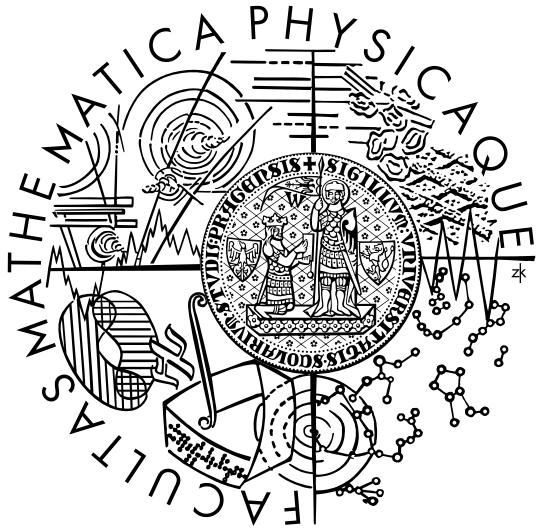
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Malohlava Michal

Using Stratego/XT for Generation of Software Connectors

Department of Software Engineering
Supervisor: RNDr. Tomáš Bureš, Ph.D.
Study Program: Computer Science, Software Systems

First at all, I would like to thank my advisor Tomáš Bureš for his valuable suggestions, observations and help with writing and also to my family and friends for their support in my life and studies.

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on December 19, 2006

Michal Malohlava

Název práce: Using Stratego/XT for Generation of Software Connectors

Autor: Michal Malohlava

Katedra: Katedra Softwarového Inženýřství

Vedoucí diplomové práce: RNDr. Tomáš Bureš, Ph.D.

e-mail vedoucího: bures@nenya.ms.mff.cuni.cz

Abstrakt: *Softwarové konektory hrají významnou roli v komponentových systémech, kde pomáhají modelovat a realizovat spojení mezi komponentami. Krom toho můžou také rozšiřovat vlastnosti spojení přidáním definované funkcionality (např. logování, monitorování, adaptace). Pro tyto účely je ale nutné konektory generovat v závislosti na podmínkách, které jsou specifikovány vlastními komponentami, prostředím a nebo návrhářem.*

Tato práce se snaží rozšířit existující generátor konektorů [33] pomocí systému Stratego/XT*, který zahrnuje jazyk pro implementaci programových transformací a sadu podpůrných nástrojů. Pomocí tohoto systému realizujeme způsob, kterým lze snadno definovat vlastní implementaci softwarového konektoru a následně vygenerovat zdrojový kód konektoru dle daných požadavků.*

Klíčová slova: Stratego/XT, softwarové konektory, generování kódu, transformace kódu, DSL

Title: Using Stratego/XT for Generation of Software Connectors

Author: Michal Malohlava

Department: Department of Software Engineering

Supervisor: RNDr. Tomáš Bureš, Ph.D.

Supervisor's e-mail address: bures@nenya.ms.mff.cuni.cz

Abstract: *Software connectors are used in component based systems as a special entities modeling and realizing component interactions. Besides this behavior, connectors can provide extra functionality and benefits (e.g. logging, adaptation, monitoring). This approach requires generation of connector code with respect to requirements of components, a target environment and features specified at the design stage.*

In this thesis we show how to extend the existing connector generator [33] by the Stratego/XT *transformation engine, which includes a language for implementing program transformations and a collection of transformation tools. We use the toolset to realize a simple method of defining connector implementation, which is use as a template for a process of generation source code.*

Keywords: Stratego/XT, software connectors, code generation, program transformations, DSL

Contents

Chapter 1

Introduction and motivation

The computers and software have become an important part of today world. They affect production and everyday activities of many people. Therefore computers, software and also a process of their preparation are in the centre of improving. The part of engineering, which is interested in software, its design, development and meliorating, is called software engineering. It is a young discipline, and is still developing. It has several directions in which it is converging. One paradigm is a component based software engineering (CBSE). This is a fast evolving sphere, which provides a way to build large software systems with a top-down method by dividing them into small parts called components. Each component expresses some functionality which is exactly specified. Another important part of CBSE is collaboration of components, which is performed via well defined interfaces. Because the component is an independent part of a system it can be reused in different applications. These benefits can lead to faster development, more robust and highly scalable applications with easy-to-maintain code bases.

A lot of companies has developed their proprietary component systems. Microsoft has a family of component models which includes COM+, DCOM and the newest .NET component model [3]. Sun Microsystems offers popular Enterprise Java Beans (EJB [8]), which provide many features. We have to also mention CCM (Corba Component Model [4]) developed by Object Management Group. Except these closed and proprietary systems several academic systems exist (Fractal [10], SOFA [17], ArchJava [2]).

Each of these systems understand components and their interconnecting in its own manner. But, what is the component in fact?

From a developer point of view a component is a program entity which includes only a business logic and it should not be concerned in a communication. The component just needs to communicate with other components via well-defined interfaces. On the other hand the designers of component systems need a way to model, represent and specify properties of a connection between components. Often

we need also to monitor or adapt communication of components at run-time. For these purposes software connectors seems to be most eligible.

These first-class entities (see *Figure 1.1*) realize interaction-specific tasks for components. They can mediate various communication styles (method call, messaging, streaming - [41]), bypass differences between component systems and add extra-value to a connection (e.g. logging, introspection, measurement, adaptation). The communication layer provided by a connector is simply configurable and can be individually prepared for each binding between components. Hence the connector separates a communication logic from the business logic of the component.

Connectors also help us in connecting heterogeneous systems, which are based on different types of components. A majority of component systems provide a communication only among components of the same component model and a combination of multiple component systems in one application brings a lot of problems, because each component system supporting distribution uses a native middleware (e.g. EJB use RMI, CCM uses CORBA, .NET uses .NET Remoting). Thus there is a demand to cross the differences (e.g. communication styles, interfaces) in middleware layers. Several bespoke solutions of the problem exist, but they provide only connecting of chosen component systems - e.g middleware bridges *J-Integra* [12], *JNBridge* [14] mediating a connection between EJB and .NET components. But these solutions are often closed and they are not scalable.

Thus connecting components via connectors seems as a good idea which could bring a lot of benefits for CBSE.

There are two ways of implementing connectors. This first method implements connectors separately before deployment of components. This needs that the connector for each type of the component system and binding should be prepared. But this way does not support adaptation of the connector at the deployment time when specific requirements (monitoring, security) can occur.

The second method moves a generation of connectors to a deployment stage. This decision offers a modification of connector code at deployment time, which can often produce more optimized target code. Accordingly the generation of connectors seems as the best way of connector preparation, but on the other side, the connector generation process itself must not slow down a deployment process significantly.

The idea of connector generation is not new. There are already several attempts of connector generation. The simple generators works on the principle of generation of stubs and skeletons (e.g. CORBA [23], RMI [20]). But this method does not offer additional adjustment of code generation. More advanced solutions work with a template system and provide an abstract definition of a connector implementation. One representative is *Openwings* system [16], which realizes connector generation. The connector in Openwings view is a bundle of Java classes, which are implemented for the given binding between components. At run-time they are

loaded on demand and used for connecting components. The main problem of this concept (which is not only specific for Openwings) is impossibility of connector adaptation for specific demands on deployment time, because the connector is already compiled into a Java class.

Here we also have to mention a connector generator [38]. It is well designed and suitable for connector generation at deployment time. But the problem of this generator is a definition of a connector implementation, which relies on a Java class. The class produces a target code of the connector itself. This feature makes the generator not easy usable and error prone, because the developer has to write new Java class producing the implementation of a new connector. The concept bears a lot of extra code, which is not really necessary for the output connector implementation.

Therefore we want to improve the definition of a connector implementation in the existing generator and relocate the definition from Java code into *code template*.

In our view we would like to specify an abstract definition of the connector implementation at design time. Then it would be completed and adjusted at deployment time and a target connector would be prepared for connecting given components. This idea needs a way to write an abstract definition of a connector implementation, which can be easily transformed into a target code. The definition of the connector implementation should be simple human readable and writable by hand. It should also allow generation of different target code and permit code reuse. These demands lead us to build a definition of the connector implementation on some well-known programming language (e.g. Java), which will be modified to support source code transformations. The transformation of the proposed language into a target language should be fast and easy definable.

The main idea of our concept is using source code transformations which transmute an input written in one language into a program implemented in another language. The input language will be designed to agree with our requirements and the target language will be Java. Thus transformation tool should also allow definition of a source and target language grammar.

Nowadays many source code transformers (e.g XSLT [9], JET [6]) exist, but we want to aim at a leading representative, which is Stratego/XT. It complies with all our requirements and also provides a set of supporting tools, which can help us to facilitate source code transformations.

Thus general goals of this thesis are:

(i) design of a new domain specific language for defining connector implementation

(ii) proposal and implementation of a method for transforming defined connector implementation into a chosen target language

(iii) integration of the existing connector generator [38] with STRATEGO language. The resulting generator should be faster and mainly easy usable then the existing solution of connector generation.

(iv) testing an eligibility of the STRATEGO language for source code transformations and generation in this context.

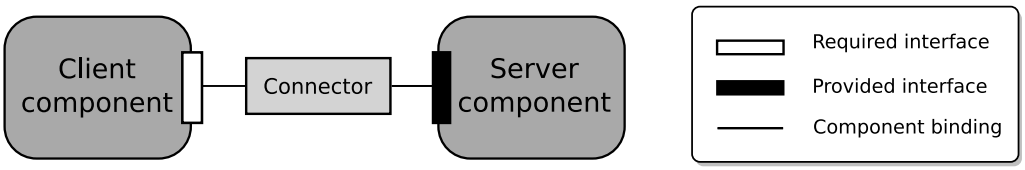


Figure 1.1: An example of a connector

1.1 Structure of the text

Following chapters start with an introduction into a connector model. The *Chapter 2* also shows the existing connector generator. The STRATEGO language and supporting tools are shortly described in the *Chapter 3*. The *Chapter 4* revisits the goals of the thesis and analyzes them deeply. The architecture of the proposed connector generator is shown in the *Chapter 5*. Following *Chapter 6* describes features and implementation of language for defining connectors. Evaluation of the proposed solutions is done in *Chapter 7* and the next *Chapter 8* discusses related work. The last *Chapter 9* concludes the thesis and gives possible ideas for future work.

Chapter 2

Connectors and their generation

Software connectors (see *Figure 2.1*) are first-class entities providing a configurable communication layer. They connect different components, which can live on various computers. Hence connectors span different address spaces.

2.1 Connector model

Connectors are described by a connector model. Numbers of connector models exist and each of them provides a different set of features and a different connector structure. This work is using a connector model presented in [33], [30] based on the model [38], which was changed to suit the automatic connector generation.

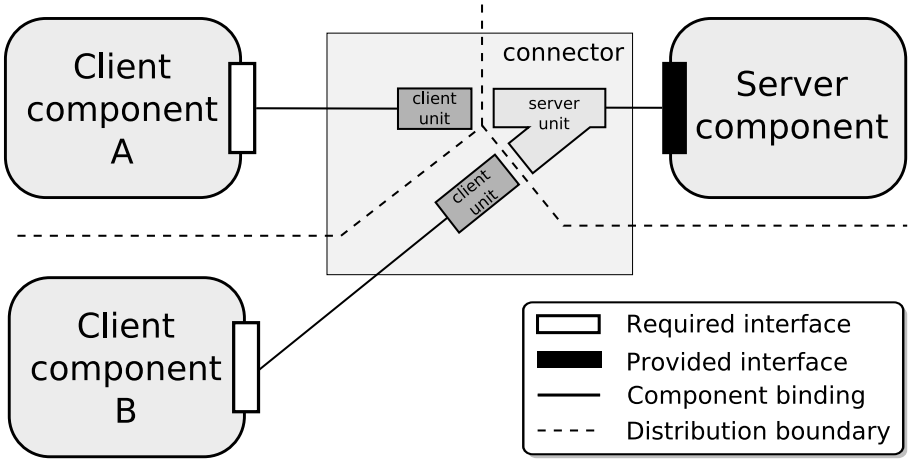


Figure 2.1: A simple view of a connector architecture

2.2 Generation process

A process of the generation of connectors is a part of a deployment manager. The work [31] describes a deployment process based on OMG D&C [24]. OMG D&C defines a deployment of a homogeneous application in a platform independent way. It was extended to support a deployment of components from different component models by introducing software connectors like entities responsible for components interactions. The OMG specification was also extended to support connector construction during deployment time when we have enough information to prepare a connector implementation suiting deployed components. The paper [32] describes an automated process of a connector generation based on a high-level connector specification. The generation also allows a connector adaptation for a particular target deployment environment.

The generation process itself can be divided into two stages (see *Figure 2.2*). Each stage expects a description of a connector in a particular level of abstraction and produces a more specific description. The input of the generator is a high-level connector specification, which prescribes basic requirements. It is partly written by a connector designer and completed by a tool, which is concerned with component deployment. This description is transformed into source code. Between the input connector description and the result code is a big semantic gap, which is filled by an intermediate connector description called *low-level connector configuration*.

The first stage of the connector generation is responsible for resolving a connector architecture. It processes an input high-level connector descriptor and tries to find a satisfactory connector configuration. The result is used as an input for the second stage, which generates target source code of the connector with help of code templates.

2.2.1 High-level connector specification

The input of the connector generator is a high-level specification of a connector. This descriptor allows a user to specify a required connector at the design stage in a way which is convenient to human. The specification is based on a communication style and non-functional properties (e.g. security, monitoring).

The communication style is determined by a component designer and it is associated with the given component interface. Some NFPs (e.g. security requirements) can be also connected with a component interface at the design stage. The additional NFPs (e.g. monitoring of calls on interface) are prescribed by a deployer tool at assembly and deployment time.

The NFP is a set of named attributes specified in a dot notation (see *Listing 2.1*). A description of required NFPs is written as a restriction over values of

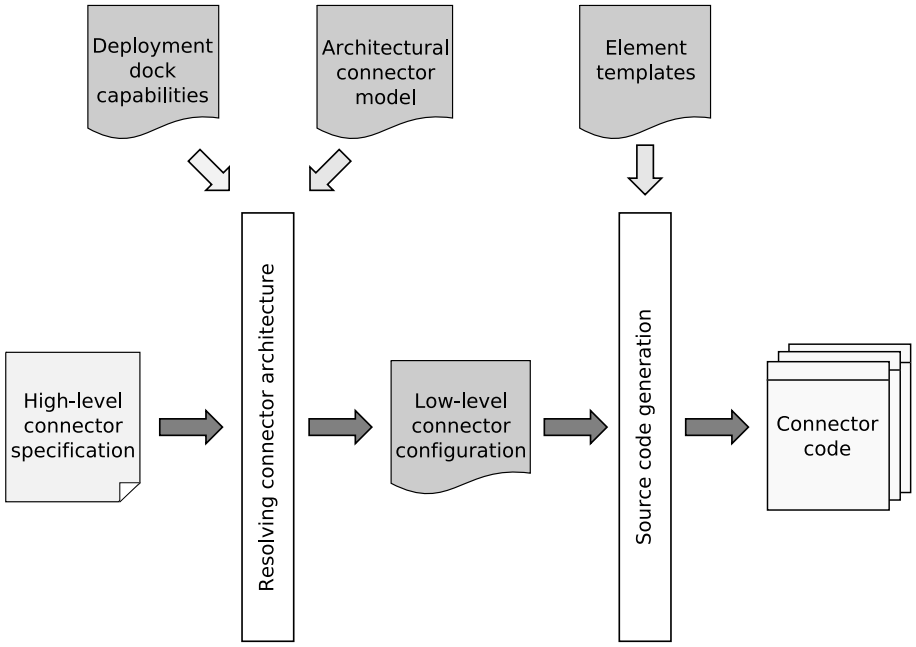


Figure 2.2: Overview of the generation process

specified NFP attributes. For this purpose the model uses a predicate over chosen NFP attributes. An example of such specification is shown in *Listing 2.1.*

```
(security.connection.type='ssh' && security.server.cert='server.cert')
```

Listing 2.1: An example of a predicate of NFPs

2.2.2 Low-level connector configuration

The low-level connector configuration is a structural view of a connector. It describes an extended hierarchy of inner elements called *connector elements*, their ports and bindings among them. The configuration is similar to the description of a component system - there are also nested entities corresponding to *connector elements* and bindings among them.

We can look at the low-level connector configuration as on a tree. In the root of the tree there is a representation of a connector and nodes of the tree correspond to connector elements. From one level of nesting view the connector is constituted of connectors units. Because the connector is a distribution entity we can identify the largest parts of the connector which can independently live in a deployment dock. And these parts are represented by connector units. But in fact a connector unit conforms to a connector element.

As we mentioned above the basic building entity of a connector is a *connector element*. In the low-level configuration the element specification includes *element type*, actual signatures of element ports and specification of sub-elements. The

element type is only a black-box view of the element because it specifies element ports, which play the role of element interfaces. The connector model prescribes three types of element ports:

- *provided ports*
- *required ports*
- *remote ports*

Each element port in the connector configuration has assigned an actual signature. It means that all information about a port interface is known. Between ports the bindings are constituted. We distinguish two types of bindings:

- *local binding* - this binding is between required, provided or provided, required ports. It connects elements inside one connector unit (inside one address space), hence it is implemented by local calls. It also provides a communication between a element and a component.
- *remote binding* - this binding is only between remote ports. It connects different address spaces thus it should be implemented by a middleware. The meta-model doesn't distinguish a communication style or a direction of remote bindings. An implementation of the remote bindings depends only on elements, which provide remote ports.

The element itself can have two possible implementations. It can be *primitive* or *composite*. The primitive element is presented by a code template, which implements desired functionality (e.g marshaling, logging). The primitive elements correspond to leafs in a tree view of the connector. The composite element describes nested elements, bindings between them and assembly of sub-elements. It conforms to inner nodes in a hierarchy view of the connector.

Elements inside one connector unit are connected only via local bindings, because they live in one address space. If the element has a remote port it has to be delegated to a port of the bounding element.

On the other hand connector units represents entities which are typically located in different address spaces. Therefore a connector unit communicates on one side locally with a component and on the other side remotely with another connector unit. Remote communication is provided by some type of a middleware (RMI [20], JMS [21], JavaSpaces [22], ...) which depends on a specialization of the connector.

Figure 2.3 shows a simple connector configuration. It consists of a client unit and one server unit. The client unit is implemented by a client unit element. It is a composite element and includes two nesting elements. The first one is a *logger*

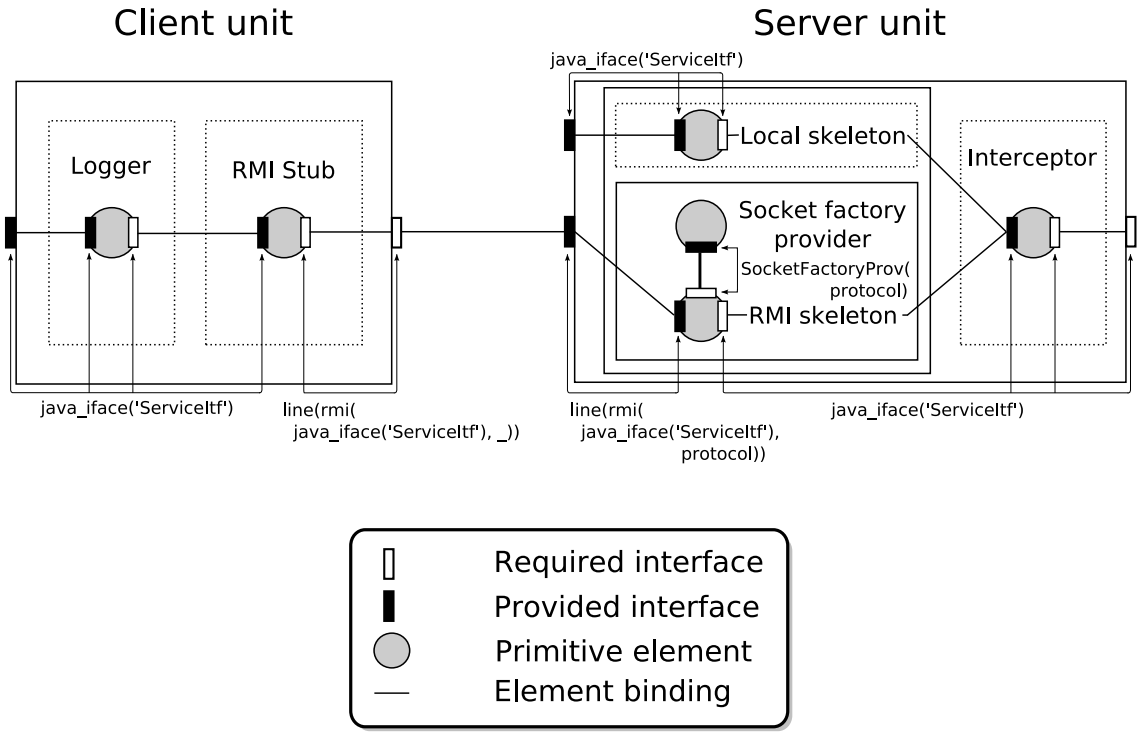


Figure 2.3: An example of a low-level connector configuration

primitive element, which is a special type of an interceptor. The second is a *stub* element providing communication with a skeleton placed on the server side with help of the RMI middleware. The client unit is connected to the server unit via a remote binding. Its configuration includes two inner elements - a skeleton collection element, which is an example of a composite element, and an *interceptor* element, which does not affect calls and it is used e.g. for monitoring, debugging or collecting statistics. The skeleton collection element includes two nested skeleton elements. The first one is a simple *local skeleton* capturing calls on a local interface. The second element is a composite element which includes a *RMI skeleton* and a *socket factory provider*. The RMI skeleton uses the factory provider element to obtain a socket factory for the given *protocol*. Multiple skeleton elements permit the server unit to serve multiple clients implementing different types of a middleware. The example shows that the low-level configuration of a connector includes all port interfaces with a resolved signature.

2.2.3 Resolving connector architecture

The first step of a connector generation process is resolving a connector architecture. The input for this part of the generator is a high-level connector specification described in *Subsection 2.2.1 High-level connector specification*. The resolver translates the specification into a low-level connector configuration. During resolving it uses information about a target deployment environment, where components are

situated, and also an architectural connector model (it is only one-level of nesting view of a connector - see *Figure 2.4* [1]).

The process of resolving an architecture searches the space of all possible configurations and tries to find appropriate configurations suitable to the specified high-level connector specification and deployment requirements. The process of resolving a connector architecture first chooses a connector architecture of a root element and then recursively assigns architectures to connector units and sub-elements. Among found configurations one is selected as the best in according to the "cost" of the connector configuration.

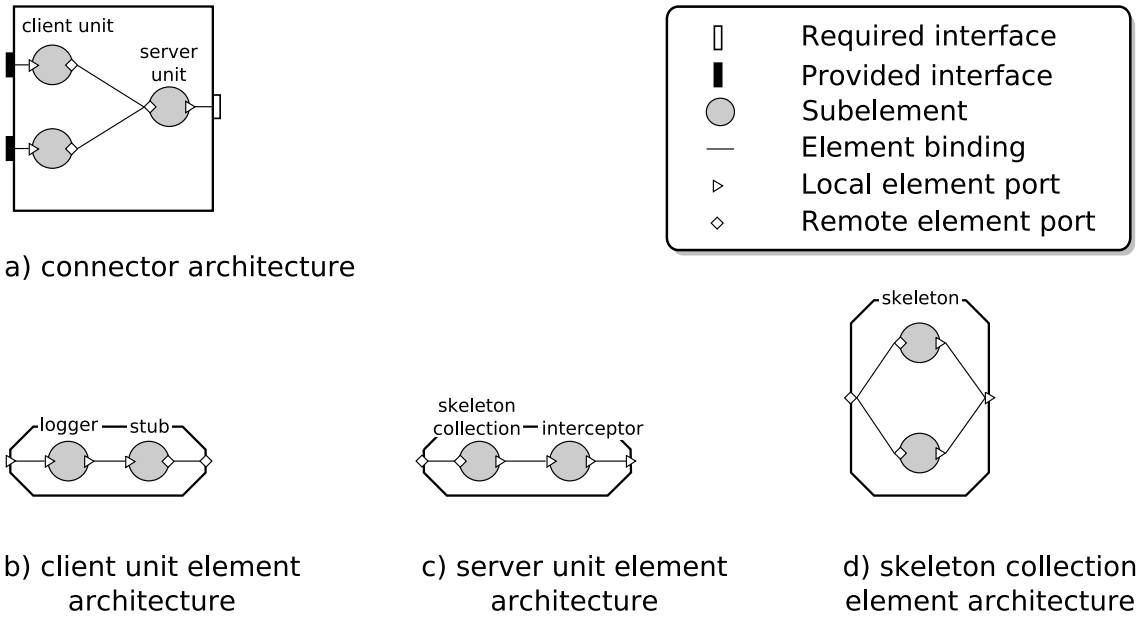


Figure 2.4: A simplified connector architecture on the one level of nesting

2.2.4 Source code generation

Source code generation is based on adapting a connector element template with help of an input low-level connector configuration. It is just deterministic process, which

[1] The connector architectural model is not described here, because the connector element generator is just interested in the connector low-level configuration. But at least briefly, the connector element architecture is a gray-box view of the element. It describes element's implementation (primitive, composite), but it is not concerned with specifying element ports. For this task the *connector type* is responsible. Hence there can be a number of architectures for an element type. The *Figure 2.4a* shows a simple connector architecture. It includes multiple client units and one server unit. The architecture of a client unit element is shown on *Figure 2.4b*. It is a composite element containing two primitive elements - a logger and a stub. The client unit is connected to the server unit (*Figure 2.4c*), which includes a skeleton collection element and an interceptor. The collection element (*Figure 2.4d*) have multiple skeletons and each of them can implement different type of middleware. Hence server unit can support multiple client units, each using different middleware.

For further details of the connector architecture model the work [33] is recommended.

transforms a template code into a target source code.

The concept of connector units is important here. Because the unit is standalone part of a connector, which can be instantiated separately, thus the generator produces code of connector units. But we have mentioned the connector unit corresponds to a connector element. Therefore the generator is only interested in generation of connector elements.

For primitive connector element the generator just adapts an element code template to particular interfaces. In case of a composite element it has to also produce builder code, which constructs sub-elements and bindings among them.

2.3 Existing generator

In our work we will extend the existing connector generator presented by [33]. The generator itself is divided into two parts which provide processes described above. A part corresponding to resolving a connector architecture is called the *architecture resolver* and code generation is the responsibility of the *element generator*. The orchestration of these parts is controlled by a generation manager, which is represented by Java class `GenerationManager`.

2.3.1 Architecture resolver

The architecture resolver is representation by the Java class `ArchResolver`. The logic of resolving the best connector architecture is implemented in Prolog ([33], [32]), because it provides backtracking naturally, which is the main idea of finding the right architecture. A knowledge base of Prolog is filled with information about the connector and element architectures. Then Prolog is used for finding the best solution of the connector architecture with backtracking the search tree of all possible connector configurations. The resolved architecture contains all required information for code generation.

The input high-level connector specification is defined in the XML language (see an example on *Listing 2.2*) and it is produced by a deployment tool, which put together requirements on deployment docks and NFPs. The example shows a high-level specification of a connector with the name `client_unit` containing two connector units. One is situated on a client side and it provides logging service and the second one is a server unit.

The architecture resolver needs also a specification of possible element architectures with their costs. The example of such specification is shown on *Listing 2.3*. The file also includes allowable combinations of NFP names and attributes.

```
  <unit name="client_unit" dock="node1">
    <nfp-requirement predicate="nfp_mapping(Unit,'communication_style','method_invocation')"/>
    <nfp-requirement predicate="nfp_mapping(Unit,'logging',_)"/>
    <port name="call" type="provided" signature="java_interface('test.TestIface')"/>
  </unit>

  <unit name="server_unit" dock="node1">
    <nfp-requirement predicate="nfp_mapping(Unit,'communication_style','method_invocation')"/>
    <port name="call" type="required" signature="java_interface('test.TestIface')"/>
    <nfp-requirement predicate="nfp_mapping(Unit,'logging',_)"/>
  </unit>
</specification>
```

Listing 2.2: An example of a high-level connector specification produced by a deployer tool

```
<?xml version="1.0" encoding="UTF-8" ?>

<element name="logged_client_unit" type="rpc_client_unit" impl-class="LoggedClientUnit">

  <architecture cost="0">
    <inst name="logger" type="logger"/>
    <inst name="stub" type="stub"/>
    <binding port1="call" element2="logger" port2="in"/>
    <binding element1="logger" port1="out" element2="stub" port2="call"/>
    <binding element1="stub" port1="line" port2="line"/>
  </architecture>

  <nfp-declarations>
    <nfp-mapping name="logging" value="Value">
      get_elem(This,'logger',SE_Logger),
      nfp_mapping(SE_Logger,'logging',Value)
    </nfp-mapping>
  </nfp-declarations>

  ...

  </script>

</element>
```

Listing 2.3: An example of an element architecture descriptor

2.3.2 Element generator

The connector configuration produced by the architecture resolver selects an element implementation and prescribes to which interfaces the element should be adapted. The element generator adapts element templates, generates their source code (e.g. Java code) and builds the whole connector units. The generator itself is implemented in the Java code by the class `ElementGenerator`. It controls source code generation, its compilation and post modification via performing defined actions. The process of an element generation is described by a script (see *Listing 2.4* containing build commands. Each command corresponds to a Java class implementing interface `ActionInterface` and is interested in one part of an element production.

The command which realizes source code generation is called *jimpl* and is implemented by the Java class `JImpl`. It takes as an input the class name of a code generator and the name of a template representing element implementation.

The code generator is a Java class implementing the interface `JImplGeneratorInterface` and it is responsible for template transformation into a target language. Code generators constitute a hierarchy shown on *Figure 2.5*. There are two general implementations for primitive elements (class `PrimitiveGenerator`) and for composite elements (class `CompositeGenerator`). These classes can be specialized (by inheritance) to satisfy special template element demands (e.g. classes `ConsoleLog`, `LocalStub`).

The template itself represents only a static part of an element implementation and the dynamic part is produced by a code generator class mentioned above. It is written in a Java language and contains tags enclosed in `%`. These tags are expanded by the given code generator.

The generated source code is compiled by the command *javac* which calls a java native compiler. Then classes can be post-processed by various commands - e.g. *rmic* which generates RMI stubs and skeletons[2]. The action *delete* is responsible for cleaning.

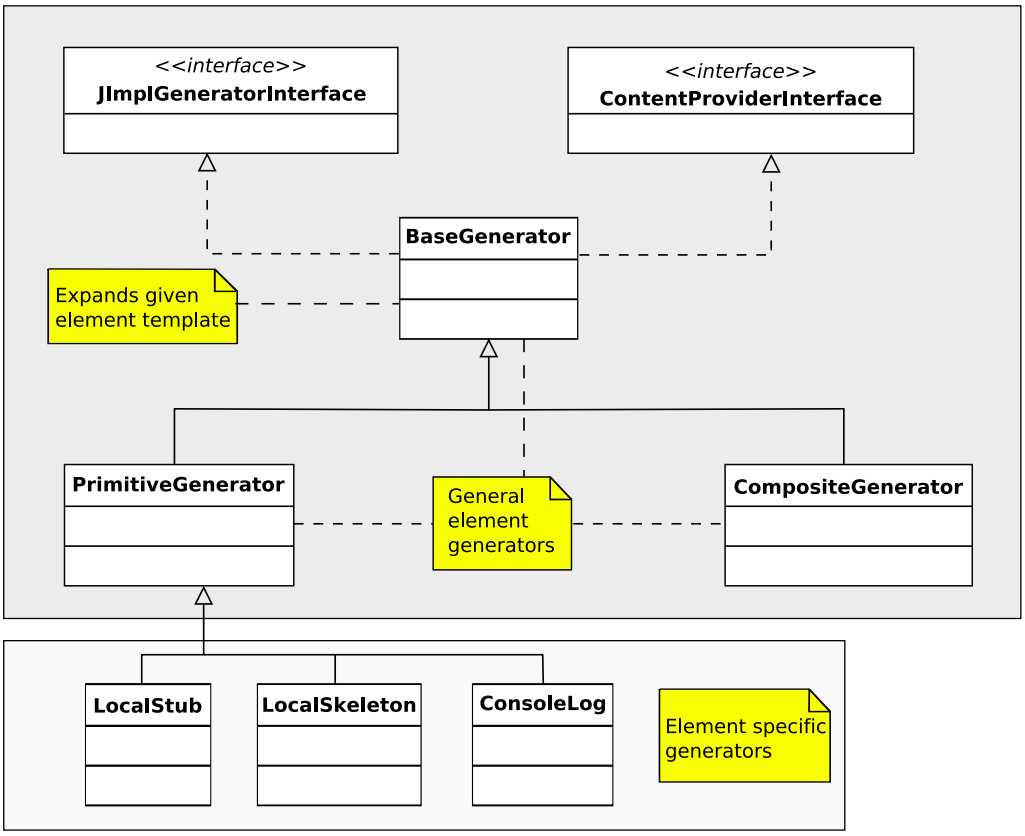


Figure 2.5: UML diagram of the hierarchy of template expanders

[2]Explicit calling *rmic* is mandatory only in the case of using Java 1.4

```
<?xml version="1.0" encoding="UTF-8" ?>

<element name="logged_client_unit" type="rpc_client_unit" impl-class="LoggedClientUnit">
   ...
  <script>
    <command action="jimpl">
      <param name="generator" value="org...generators.CompositeGenerator"/>
      <param name="class" value="LoggedClientUnit"/>
      <param name="template" value="compound_default.template" />
    </command>

    <command action="javac">
      <param name="class" value="LoggedClientUnit"/>
    </command>

    <command action="delete">
      <param name="source" value="LoggedClientUnit"/>
    </command>

  </script>

</element>
```

Listing 2.4: An example of building script of a simple element

```
package %PACKAGE%;

imports org.objectweb.dsrg.deployment.connector.runtime.*;

public class %CLASS% implements
    ElementLocalServer,
    ElementLocalClient,
    ElementRemoteServer,
    ElementRemoteClient {

      protected Element[] subElements;
      protected UnitReferenceBundle[] boundedToRemoteRef;

      public %CLASS%() {
      }

      %INIT_METHODS%
}
```

Listing 2.5: An example of a template for a composite element

Chapter 3

Overview of Stratego/XT

Stratego/XT [18] is a combination of the Stratego programming language with the XT bundle of transformation tools.

XT is a set of useful transformation tools, which are used for generation of parsers, pretty printing, abstract syntax tree transforming, building and bundling of systems. The package is based on the Syntax Definition Formalism (SDF), the Generic Pretty-Printing (GPP) package and the Stratego language.

Stratego is a special purpose transformation language, which is based on the term rewriting and can be seen as the implementation of the strategic programming paradigm. It provides numerous features such as grammar variables, concrete object syntax and dynamic rules that make it very suitable for implementing program transformations.

3.1 Architecture

The XT package is composed of different components. Each component is an executable program, which can be used directly from a command line. A set of components can be combined via a shell pipe. An example of some typical pipeline is shown on the *Figure 3.1.* A source program is parsed by a parser based on the SDF grammar definition. The result of this step is an abstract syntax tree which is then transformed by the collection of transformation tools (e.g. a desugarer, an optimizer, a simplifier, a template extender - all of them are written in the Stratego language). Finally the result can be processed by a pretty-printer which produces source code in a target language. The target language can be the same as an input language or can be different in case of language translation.

The grammar definition of a language plays the central role in Stratego/XT. It uses a SDF language for describing grammars (see *Section 3.2 Syntax Definition Formalism SDF*).

XT tools exchange a structured representation of a program - an abstract syntax tree (AST). There is one-to-one correspondence between trees and prefix terms. A term is a constructor applied to zero or more terms. Strings and integers are also terms - e.g `Plus(Int("3"), Int("4"))` is a representation of `3 + 4`. In STRATEGO/XT tools use *ATerms* (Annotated Terms) like internal as well as external representation of programs. ATerm format is discussed later in *Section 3.3 ATerm*.

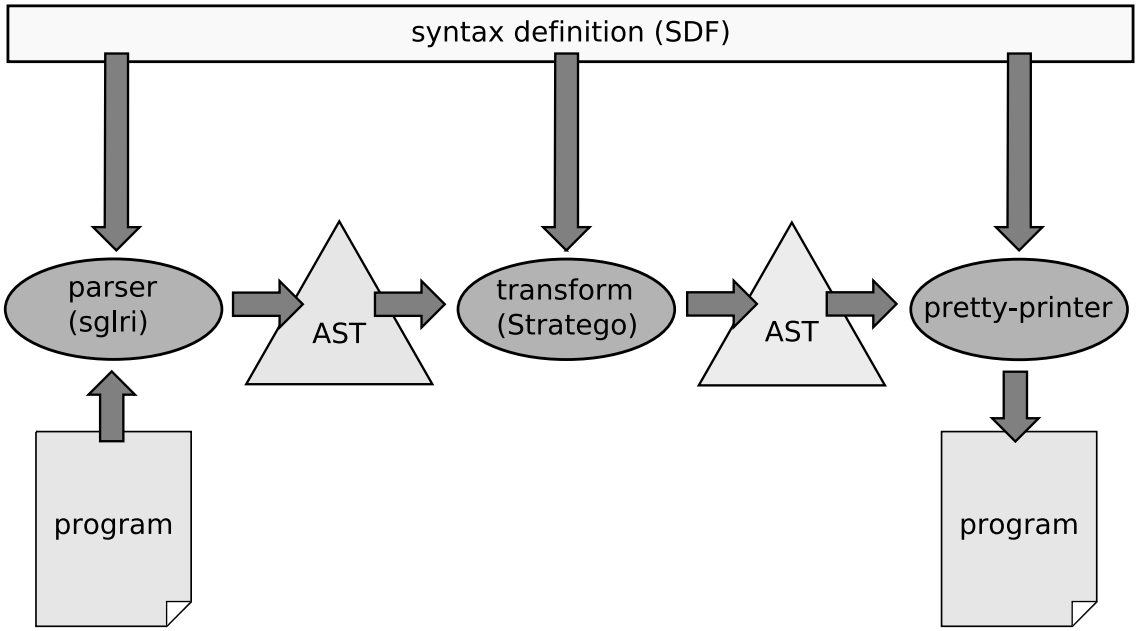


Figure 3.1: A structure of the STRATEGO/XT program transformation system

3.2 Syntax Definition Formalism SDF

SDF is a language for defining syntax of programming languages. It provides several features which permit writing syntax definitions simply. First contribution is a modular syntax definition. Syntax can be split into several modules and they can be reused in different syntax definitions. Second, a lexical and context-free syntax are integrated in a single formalism in which the complete syntax of a language can be defined (using Scannerless generalized LR parsing [35]). Third, SDF includes declarative disambiguation constructs (like priorities, reject productions and follow restrictions) which support deciding for the right abstract syntax tree.

SDF belongs to a set of declarative languages. This means that the syntax definition can be used for different purposes: a generation of a parser, pretty-printers and data type definitions.

A parser is generated automatically from the syntax definition in SDF. The resulting parser is based on *Scannerless Generalized-LR parsing* ([35]) and it produces

ATerm representation of an input program.

3.2.1 Abstract syntax tree

An abstract syntax tree (AST) is a tree representation of a source program. Internal nodes are labeled by operators and leaf nodes represent the operands (i.e. nullary operators like constants, variables). It is derived from a syntax tree (ST), but in comparison with it AST does not contains redundant layout information and comments - i.e. information which do not affect semantics of the input program. AST is used like an internal representation of programs in parsers and also in the STRATEGO/XT system. STRATEGO itself stores AST in a general term format called ATerm, which is described in *Section 3.3 ATerm*.

3.2.2 Modular structure

SDF specification can include several modules. Each module can defines syntax rules (also called functions, derivation rules). All entities defined in the module may be visible or invisible in other modules. The module can import another module and all declarations of the imported module become accessible in the importing module. And additionally they become exported by the importing module.

```
module Expr

imports
    Expr-expressions
    Expr-priorities
    Expr-layout

exports
    context-free start-symbols  Exp
```

Listing 3.1: An example of a simple module

3.2.3 Symbols and productions

Each syntax rule consists of "symbols". Symbols are similar to terminals and non-terminals in other grammar definitions formalism (e.g. Backus-Naur form, BNF). The elementary symbols are:

- *sort* corresponds to non-terminals, e.g. `Exp`
- *literal* corresponds to terminals, e.g. `"+"`
- *character classes* corresponds to set of characters e.g. `[a-z]`

These elementary symbols create more complex expressions via operators:

- *option* - the postfix operator `?` defines an optional part in the syntax rule. E.g. `PackageSection?` defines zero or exactly one occurrence of `PackageSection`.
- *sequence* - the operator `(...)` allows grouping of two or more symbols. E.g. `("import" Id)`.
- *repetition* - the repetition operator expresses that a symbol can occur several times. It also expresses minimal number of repetitions - at least zero times (`*`) or at least once (`+`).
- *alternative* - the operator `|` expresses the choice between two symbols. E.g. `Int | Bool`.
- *tuple* - the expression `(T1, T2)` creates a tuple of two terms.

A production (also called a syntax function, a syntax rule or a derivation rule) has the form $A_1 A_2 \dots A_n \rightarrow A_0$ where $A_0, \dots A_n$ are symbols. So it means, that the production takes list of symbols and produces new symbol. It is similar to derivation rules in context-free grammars: $A_0 \rightarrow A_1 A_2 \dots A_n$ where A_0 is a non-terminal and $A_1, \dots A_n$ are non-terminals and terminals. Each production can have special attributes specified in curly brackets e.g. `{cons("Int")}`. These attributes can solve disambiguation (see *Subsection 3.2.7 Solving ambiguity*) or modify generation of an abstract syntax tree (see *Subsubsection 3.2.5.1 Constructor attribute*)

3.2.4 Lexical syntax

A lexical syntax describes a low level structure of a language. It divides an input text into *lexical tokens*. A lexical token consists of a sort name and own text of the token. The lexical syntax also specifies which part of a text will be skipped (e.g. layout symbols, comments).

A description of the lexical syntax contains a set of *lexical functions*. Each consists of a regular expression on the left side and a result sort on the right side separated by `->`.

```
module Expr-literals

exports
  sorts Int

  lexical syntax
    "0"             -> Int
    [1-9][0-9]*     -> Int

  lexical restrictions
    Int     -/- [0-9]
```

Listing 3.2: A sample of a lexical syntax definition

3.2.5 Context-free syntax

The context-free syntax describes a structure of sentences in a language. A declaration of a rule consists of zero or more symbols followed by `->` and a result symbol. The result symbol may be followed by attributes which define associativity or influence a rewrite process.

Elements on the left side of the rule are separated by the invisible non-terminal `LAYOUT?` (optional `LAYOUT`) in order to permit a layout between these members. This non-terminal is automatically inserted.

3.2.5.1 Constructor attribute

A constructor attribute `cons` does not affect the syntax itself. The constructor only serves to specify the name of a node in the abstract syntax tree which is created when the syntax function associated with the constructor is applied. E.g.

```
Exp "+" Exp  -> Exp {cons("Add")}
```

```
module Expr-expressions

imports Expr-literals

exports
  sorts Exp

  context-free syntax
    Int             -> Exp {cons("Int")}
    Exp "+" Exp     -> Exp {cons("Add"), assoc}
    Exp "-" Exp     -> Exp {cons("Sub"), left}
    Exp "*" Exp     -> Exp {cons("Mul"), assoc}
    Exp "/" Exp     -> Exp {cons("Div"), assoc}
    Exp "^" Exp     -> Exp {cons("Pow"), right}

    "(" Exp ")"     -> Exp {bracket}
```

Listing 3.3: A sample of a context-free syntax definition

3.2.6 Start symbol

A context-free start symbol serves as a start symbol for the process of parsing an input program. A module can define multiple start symbols but then we have to specify before parsing which one will be used.

3.2.7 Solving ambiguity

3.2.7.1 Generalized parsing

Parsing of the input language is based on a generalized LR parser. This means that the parser finds all possible derivations for a certain input sentence. Thus result for an input string can be one syntax tree but even forest of syntax trees. For avoiding and rejecting trees SDF provides special constructs:

- prefer, avoid, reject attributes
- priorities
- associative functions
- restrictions

3.2.7.2 Prefer, avoid, reject attributes

SDF offers specifying of an attribute for each lexical even context-free function. The most important attributes for solving disambiguation are:

- *prefer* is used to indicate that a marked function should always be preferred over other functions.
- *avoid* indicates that a marked function should be used as a last resort.
- *reject* can be used to deny given construct. E.g. `"begin" -> Id {reject}` denies using the `begin` keyword as the name for an identifier.

3.2.7.3 Priorities

Relative priorities for two functions can be defined in the section `context-free priorities` with the statement `F > G`, where `F` and `G` are context-free syntax functions. This declaration says that tree nodes corresponding to the function with lower priority should occur in higher levels in the syntax tree then nodes of the function with higher priority.

3.2.7.4 Associative functions

Associative functions are productions marked by special attributes solving associativity. The attributes are often used for marking functions in the form `S op S -> S`, where we have to specify an interpretation of sentences like `S op S op S`.

Currently SDF offers four attributes:

- *left* means left associativity of the rule
- *right* expresses right associativity
- *assoc* the same sense as *left*
- *non-assoc* a marked function is not associativity. It denies expressions like `1 - 2 - 3`

3.2.7.5 Restrictions

A restrictions limits *look-ahead* for the given symbol. It expresses that a symbol cannot be followed by a character from a given character class. SDF allows writing a definition of restrictions separately for the lexical syntax and the context-free syntax. E.g. in the example *Listing 3.2 Lexical syntax* the symbol `Int` may not be followed by any numeral.

3.3 ATerm

Every program (i.e. an input sentence) is represented by terms called *ATerms*. The *ATerm* (Annotated Term) format is used for exchanging program representations between tools in the Stratego/XT package. A parser generated from the given SDF grammar produces ATerms which are processed by Stratego programs. ATerms are also used internally by Stratego programs for storing data. ATerm format uses a method of maximal sharing - it means that any term is represented only one. Other occurrences of the same term are represented by pointers to the same location.

The ATerm format provides a set of constructs for representing abstract syntax trees (see figure 3.2):

- *Application* - `Int(t)`, `Plus(t, t)`
- *List* - `[]`, `[t, t, t]`
- *Tuple* - `(t, t)`, `(t, t, t)`
- *Integer* - `16`
- *String* - `"cheers"`
- *Annotation* - `t{t,t,t}`

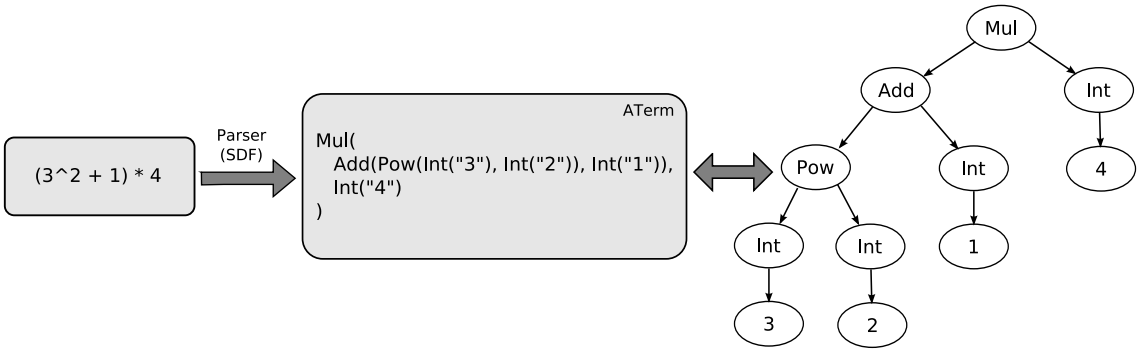


Figure 3.2: From an input sentence to AST (Abstract Syntax Tree)

3.4 Stratego programming language

STRATEGO is the language for a program transformation based on the term rewriting with definable strategies. It works with programs represented in the ATerm format (see *Section 3.3 ATerm*).

3.4.1 Stratego program representation

A STRATEGO program is divided into modules. Each module has a unique name and can import other modules. Typical program has two sections:

- *signatures* - this section contains information about terms. It declares *sorts* and *constructors* and this section is usually generated from a SDF grammar definition.
- *strategies* - this section contains a definition of strategies.
- *rules* - this section includes a definition of rewrite rules.

A selected strategy has to be specified during the compilation of a Stratego program and it will serve like starting point of a transformation process. In listing 3.4 module `EExpr` is defined. It imports the basic STRATEGO library `liblib` and another module. It also defines one strategy with name `io-EExpr` which is used for parsing input terms and producing output terms with help of the standard strategy `io-wrap`.

```
/*
 * Simple expression evaluation.
 */
module EExpr

imports liblib Expr-eval

strategies
  io-EExpr     = io-wrap(expr-eval)
```

Listing 3.4: An example of a Stratego program

3.4.2 Terms and variables

STRATEGO works with terms represented in the ATerm format (see *Section 3.3 ATerm*). To use terms in STRATEGO programs, their constructors should be declared in *signatures* section of a module. Usually these signatures are generated from a SDF definition of syntax.

A term can also contain *meta-variables*, which can be bounded to other terms. Hence a *term pattern* is a normal term with *meta-variables* - e.g. `Int(x)`.

3.4.3 Terms creating and matching

Basic operators in the STRATEGO language are term building and matching. The term building operator `!` replaces a current term with a new term pattern - e.g. `!Int("3")`. The term matching operator `?` tries to match the current term `t` to the specified term pattern `c`. The operator succeeds if there is a substitution ω of variables in `t` to subterms in the term pattern `c` such that $\omega(c) = t$ - e.g. `?Int("9")`. Matching operation also involves binding of variables in the pattern `c` to corresponding subterms of the term `t`.

3.4.4 Strategies

Strategies are used for controlling usage of rewrite rules. It can combine one or more transformations (rules, strategies) into a new transformation. A strategy definition has the form:

```
f = s
```

where `f` is a name of the strategy and `s` is a *strategy expression* - a combination of some transformations.

3.4.4.1 Basic strategies

STRATEGO presents four basic strategy operators:

- *id* - always succeeds
- *fail* - always fails
- *!p* - term creation
- *?p* - term matching

3.4.4.2 Sequencing

The sequential composition `s1 ; s2` of strategies `s1`, `s2` first applies the strategy `s1` to the current term and on the result it applies the strategy `s2`. If some of strategies `s1`, `s2` fails then the whole composed strategy fails.

3.4.4.3 Choice

STRATEGO provides several choice operators which serve for deciding between applying strategies.

The basic choice operator is a *deterministic choice* `s1 <+ s2`. It tries to apply given strategies `s1`, `s2` in that order. If the strategy `s1` succeeds then the whole choice succeeds else `s2` is applied to the original term. This strategy combiner is used in cases when we want to apply multiple mutually exclusive strategies to the term.

Another operator is a *guarded choice* `s < s1 + s2`. If the *guard* strategy `s` succeeds `s2` is applied else the strategy `s3` is applied. For example, we can define a negation strategy:

```
not(s) = s < fail + id
```

3.4.4.4 Parametrized strategies

Definitions of strategies can be parametrized by parameters:

```
f(s1, ..., sn | t1, ..., tm) = s
```

where `s1, ..., sn` are strategies and `t1, ..., tn` are terms. For example, the strategy `try(s)` which applies its argument strategy `s` and always succeeds is defined:

```
try(s) = s <+ id
```

3.4.5 Rewrite rules

Rules define one-step transformation. A named rule has the form:

```
L: p1 -> p2
```

where `L` is a rule name, `p1, p2` are term patterns. A rule defines transformation on terms. It matches an actual term and replaces it with `p2` only if `p1` matches to the

actual term. So rules definition is only syntactic sugar for the statement `?p1 !p2`.

Simple rule above is an unconditional rule, but STRATEGO support also conditional rules:

```
L: p1 -> p2 where c
```

where `L, p1, p2` has the same meaning like above and `c` is a conditional strategy. It will replace the current term with `p2` only if `p1` matches the current term and the strategy `c` succeeds.

3.4.5.1 Lambda rules

In come cases there is a need to create an anonymous rewrite rule inside some strategy expression. This rule is called the lambda rule:

```
\ p1 -> p2 where s \
```

where `p1`, `p2` are terms and `s` is a strategy.

A typical example of usage of the lambda rule is

```
map( \ x -> <mul>(x,x) \)
```

It computes the square powers of terms representing numbers, which are stored in a array term.

3.4.6 Term traversal

There exists many types of traversing trees. For some cases *bottom-up* traversing is useful, for another *top-down* is more suitable. And sometimes there is a demand to define own traversal strategy. STRATEGO supports all of them. It has built-in basic traversal strategies (*bottomup*, *topdown*, ...) but also user can define own traversal.

3.4.6.1 Congruence operators

Congruence operators provide one way of term traversing. It applies a different strategy to each argument of a specific constructor. If `c` is some term constructor and `s1, ...,sn` are strategies then `c(s1, ..., sn)` is either strategy. This strategy can be applied only to terms in the form `c(t1, ..., tn)` and it applies the strategy `s_i` to the term `t_1`. For example:

```
sadd(s) = Add(s, s)
```

shows the strategy which applies the given strategy `s` to both subterms of the `Add` term.

3.4.6.2 Generic traversal operators

STRATEGO has several basic traversal operators called *generic one-step descent operators*:

- *all(s)* - applies the strategy `s` to all direct subterms.
- *one(s)* - applies the strategy `s` to one direct subterms. It fails when application of the strategy to all subterms fails.
- *some(s)* - applies the strategy `s` to at least one direct subterm.

3.4.6.3 Standard traversal strategies

With help of generic traversal operators and congruence operators STRATEGO presents many other standard strategies:

```
bottomup(s)   = all(bottomup(s)); s
topdown(s)    = s; all(topdown(s))
innermost(s)  = bottomup(try(s; innermost(s)))
alltd(s)      = s <+ all(alltd(s))
...
```

3.4.7 Dynamic rules and their scope

Above we presented logic of strategies and rewrite rules but all these transformation are static. They are defined before program compilation and cannot be modified. But in some situation we need to create a rewrite rule in dependence on actual terms. The best example is constant folding operation in compilers of programming languages. Hence STRATEGO provides dynamic rules. Declaration of a new dynamic rule `L` has the form

```
rules(L : t1 -> t2)
```

where `L` is a name of a rule and `t1`, `t2` are terms. Thus the dynamic rule can be used anywhere in a program.

The scope of defined rule can be restricted by specific construct

```
{| L: s |}
```

where `L` is a name of the rule which should be restricted and `s` is a strategy. All dynamic rules `L` defined during execution of the strategy `s` are cleared after leaving the scope.

Chapter 4

Goals revisited

In the previous chapter we have presented the existing connector element generator. The main limitation of the solution is the code generator. It is based on a concept, where code (here the Java language is used) generates another code. The definition, what should be generated, is divided into two parts - a code template and a Java class, which provides a dynamic content of a template. The division bears on one hand a sole method how to define an universal code generator, which does not depend on a target language. On the other hand, it makes writing connectors tedious and error-prone. Splitting element template into two places also carries management of source code difficult.

We want to propose a solution, which removes inadequacies of the existing connector code generator and provides an independent layer for defining connector elements. The layer unifies the location of an element template definition and makes writing elements easier. The layer is based on a *Domain Specific Language* (DSL), which is the mixture of a target language (in our solution Java) and a meta-language. The meta-language allows accessing an element architecture description and composing target code with meta-statements. The tool which transforms the connector element language into the target language should communicate with the existing architecture solver because it provides necessary information about the element configuration.

The second important goal of this thesis is try to use a specific tool for program transformations. The tool is used for transforming a template written in a DSL to the target language (e.g. Java). In our work we test an eligibility of Stratego programming language because it is one of the most suitable tools for program transformations and also it includes additional tools for grammar definitions and pretty-printing.

Thus the main goals of this thesis are:

- design of a DSL which allows defining connector element code.

- design of the connector element code generator in Stratego language.
- the DSL and the code generator should be extensible to support new component system.
- the DSL and the code generator should be easily modifiable to add support for a new target language.
- incorporate the developed connector element code generator into the existing solution. It means to implement a new subclass of the class `BaseGenerator` which will provide communication between the Java part and the Stratego part of the connector generator.
- test eligibility of the Stratego/XT tool set for designing DSL and its transformation to the source code.

Chapter 5

Overview of proposed generator architecture

This chapter presents a summary of the connector generator architecture proposed by this master thesis. The generator is divided into two parts - the first part is an extension of the existing connector generator [33] which communicates with the second part implemented in the Stratego language.

5.1 Architecture design

The proposed generator is divided into two parts. The first part is implemented in the Java language and provides bridging between the existing architecture resolver and the Stratego part of the generator. The second part is a code generator implemented in the Stratego language and it concerns the target code generation from a connector element template and a description of an element configuration. The architecture of proposed solution is shown on *Figure 5.1.*

5.1.1 Java part of generator

The existing solution includes the architecture resolver (the class `ArchResolver`), which finds the most suitable element configuration. Information about the found element has to be passed to the Stratego part. The element configuration descriptor (element descriptor for short) has to contain a description of the element type and architecture:

- list of port descriptions. They have to specify a port name, a resolved signature of the port interface, a type of the port (provided, required, remote).
- list of sub elements in case of the composite element. Each sub element descriptor has to contain a name of the sub element on which next parts will

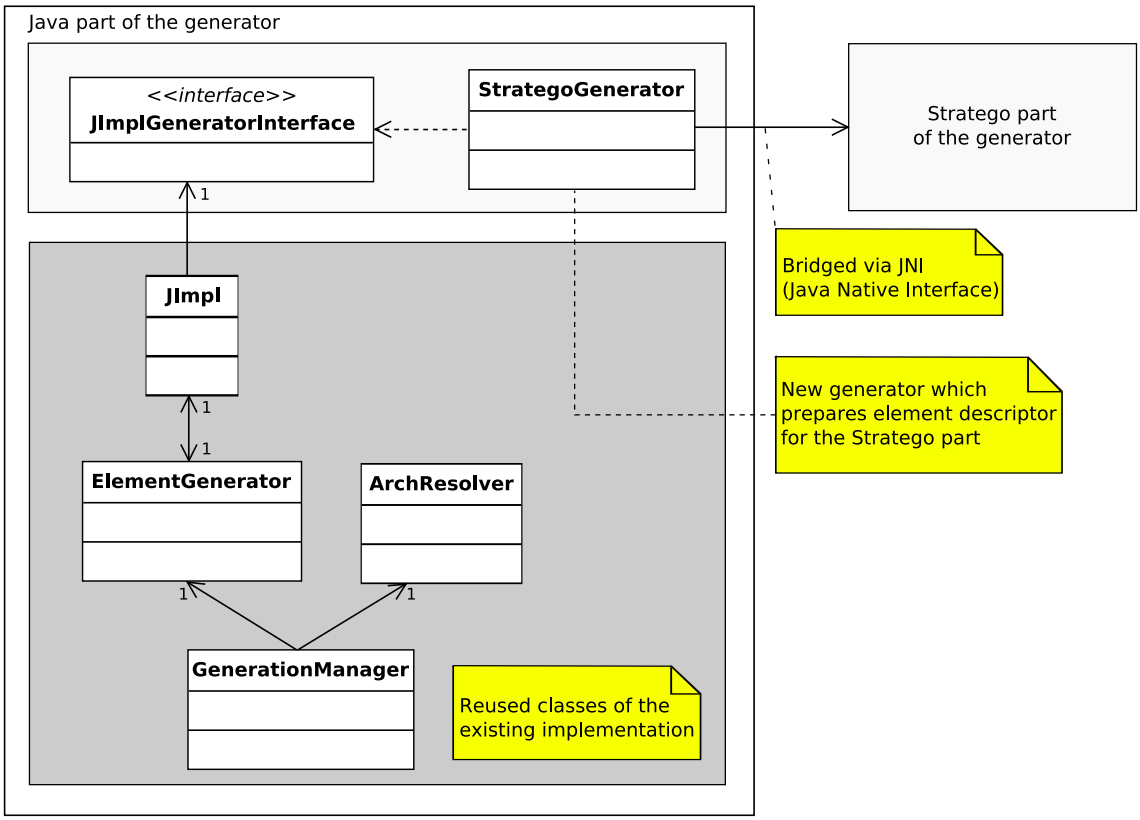


Figure 5.1: Architecture of the proposed generator

reference to, and a name of a class which implements given sub element.

- list of bindings. It contains a description of bindings between sub elements and also bindings from a bounded element to sub elements. One binding descriptor identifies a source and a target port by an element name and its port name.
- name of a template which provides connector element implementation

Except these information the STRATEGO part needs knowledge about an environment in which the element will live and also additional information like a name of the generated Java class, a name of a package.

The optimal format for exchanging the information between the Java part and STRATEGO is the XML language. It can be simply generated in Java and either STRATEGO has a tool set for XML parsing and generation.

Thus the main task of the Java part is preparation of XML source code, which describe the connector element and its environment. The existing Java implementation is extended by a new generator class `StrategoGenerator`. This class implements the interface `JImplGeneratorInterface` therefore it can be used as the plugin for the action class `JImpl`[1]. The generator class serializes a resolved element

[1]The class JImpl realizes Java source code generation with help of plugins.

represented by the class `ResolvedElementInstance` into the XML format. The produced XML descriptor has the form shown on *Listing 5.1.* Apart from descriptor generation, the class `StrategoGenerator` prepares all subelements in case of a composite element and also adapts interfaces if it is necessary.

After this preparation stage, the generator class calls the STRATEGO part with help of the class `NativeStrategoGenerator` which implements JNI (Java Native Interface - [13]) access to the STRATEGO implementation.

5.1.2 Stratego part of generator

The STRATEGO part of the generator consists of several modules (see *Figure 5.2*). Each of them has a special purpose. The architecture has a form of a pipe-line where each module takes input terms, produces output terms and the result is passed to the next module.

The first module is a XML parser which processes an input element descriptor passed from the Java part of the generator and modifies it. It is also responsible for saving the input XML fragment for later usage.

The XML descriptor of a connector element contains a name of a template which should be used for code generation. The template is found and parsed by the second module.

Then the parsed template is transformed by several stages which are described in *Section 6.4 Template evaluation*. Every stage realizes one-step transformation of the parsed template and the resulting output of the phase is then forwarded to the next stage.

In the source template there can be statements which need information from the input element descriptor. For this purpose a query part exists. It provides a uniform method how to obtain information from the stored XML descriptor and it also provides special expressions for query restrictions and simple aggregation functions.

The last part is a target code generator which takes the result of previous part, assimilates remaining meta-statements and produces pretty printed target code. The output is saved to the file with the name noted in the input description of the element.

Dividing the translation of a template to multiple stages allows us to make a majority of stages independent on a target language and displace a manipulation with the target source code into the final stage (STRATEGO module *java/**). Therefore it permits simple addition of support for a new target language only by writing a new target code generation module.

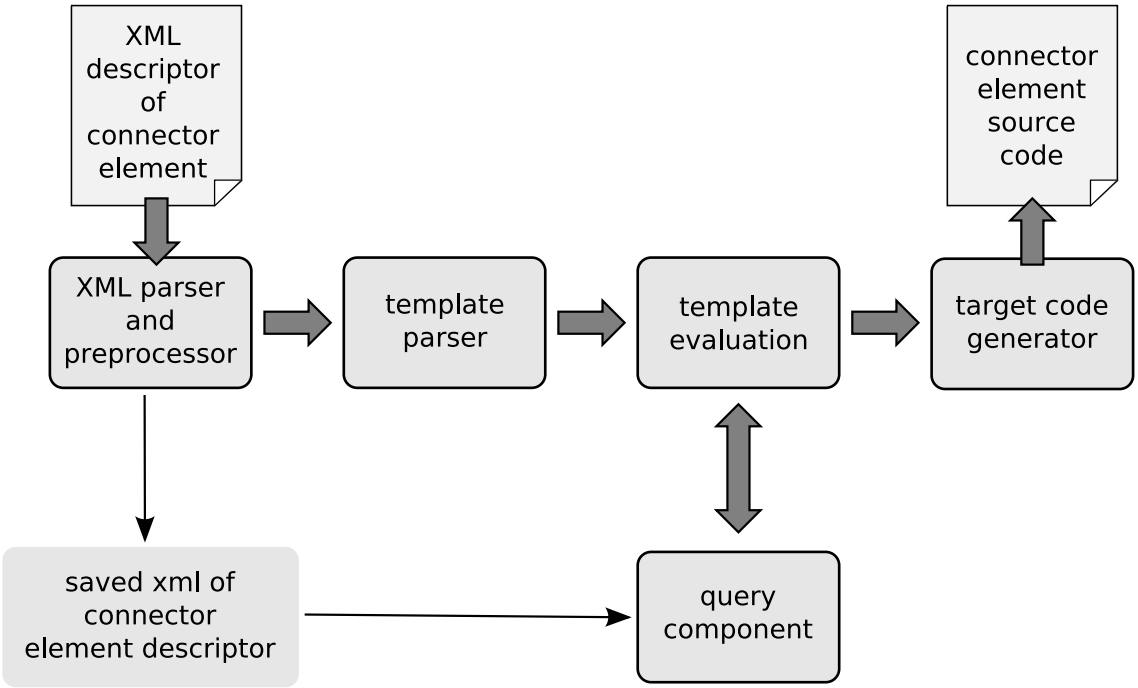


Figure 5.2: Stratego part architecture

5.1.3 Preprocessing of input XML

An input for the Stratego part is a descriptor of the element configuration stored in the XML format. The form of the XML descriptor is shown in *Listing 5.1*. This description is parsed by standard XT component `xtc-parse-xml-info` which translates XML tags into ATerm format.

Then the representation of the descriptor is modified to support easy queries through it. We want to offer a way of querying information which are not exactly specified by the stored XML fragment. For example, the query `${ports.port(name=call).boundedTo.element.name}` should return a subelement's name to which the port `call` of the current element is bounded. But the input descriptor does not contain such information and it has to be completed. Thus the preprocessor substitutes relevant information in parts where they are needed. E.g. the part `binding` contains the XML element `port` with the name of a bounded port. This element is substituted by the XML element `boundedTo` which includes the description of the given port. The port has to be also described in the input XML descriptor.

In this way modified descriptor is saved with the help of the dynamic rule `GetXML`. The rule always succeeds and rewrites the current term to the XML descriptor of the connector element:

```
save-xml = ?Document(Element(Name(_, "element"), _, desc))
    ; rules( GetXML: input -> desc )
```

Thus the element description is accessible via calling the dynamic rule `GetXML` anywhere in the STRATEGO part of the generator.

```
<element>
  <!-- template file for element -->
  <template>templates/compound_default.ellang</template>

  <!-- package of new element -->
  <package>generated.A00003</package>

  <!-- class name of generated element -->
  <classname>LoggedClientUnit</classname>

  <!-- list of ports -->
  <ports>
    <!-- description of a port -->
    <port>
      <name>call</name>
      <signature>org.congen.DemoIface</signature>
      <type>PROVIDED</type>
    </port>
    <!-- ... -->
  </ports>

  <!-- list of sub-elements -->
  <elements>
    <!-- description of a sub element -->
    <element>
      <!-- internal nam of sub element used for defining bindings -->
      <name>stub</name>
      <class>generated.A00001.LocalStub</class>
    </element>
    <!-- ... -->
  </elements>

  <!-- description of bindings -->
  <bindings>
    <binding>
      <from>
          <element>this</element>
          <port>call</port>
      </from>
      <to>
          <element>logger</element>
          <port>in</port>
      </to>
    </binding>
    <!-- ... -->
  </bindings>
</element>
```

Listing 5.1: an example of element descriptor

5.1.4 Query module

A query module is an important part of the generator. It provides a method how to retrieve specific information about the input element descriptor. A query is defined

by a statement enclosed in `${<query-string>}`. The part `query-string` contains multiple parts separated by dots. Each part of `query-string` corresponds to a name of one XML tag and whole string describes a path from the root XML element to some inner element. The query string part can also include a condition or a count operator (see below). For example, the statement:

```
${ports.port.name}
```

returns names of all described ports.

Evaluation of the query string is performed by the strategy `query`. It takes a query string like an input and tries to process it. It uses the element descriptor stored via the dynamic rule `GetXML`. Another modification of the `query` strategy is `query(|xml)` which has a term parameter `xml` and it is used instead of calling the rule `GetXML`. The general query strategy takes the first part of `query-string`, finds matching XML tag and returns its value. On the result conditions are applied or an operator is computed. Then the second part is taken and the operation is repeated with the result of the previous step. Processing of the query stops when all query parts are utilized. Final result can contains an array of text values or parts of XML descriptor. If the path described by the query string does not exist in the XML tree then the strategy fails and it does not modify the current term.

5.1.4.1 Conditions

A query string can also contain simple conditions. It has the form `tagname(name=value)` where `name` is the name of XML tag and `value` is the string value of the given tag. The example

```
${ports.port(name=call).signature}
```

returns a signature (a name of the interface) of the port with the name `call`.

5.1.4.2 Count operator

The query module also offers a simple counting operator. This operator is needed in a situation e.g. when we need to know the number of sub elements. The operator has the form `tagname#count` where `tagname` is a part of the query string and it counts the number of query results. For example

```
${elements.element#count}
```

counts the number of sub elements.

Chapter 6

Template language

We proposed a template language for defining an element implementation. The language should allow easy specifying connector element implementations and also it should be simply modifiable to support a new target language or a component system. The language should also permit a way of accessing an element architecture, which is considered as external information.

Thus we decided to make the template language as a mixture of two languages - *ElLang* language designed by us and a target language of the connector generator. In the context of this work the Java programming language was chosen. We employed a *MetaBorg* method which is provided by Stratego/XT and it offers a way to simply combine two or more languages (see *Section 6.1 Description of MetaBorg method*). The resulting template language is called ElLang-J.

6.1 Description of MetaBorg method

MetaBorg ([29], [28]) is a general method of designing new domain specific languages. The resulting language is composed of a host language in which another language is embedded. The embedding process requires a syntax definition for the host language, a syntax definition for the embedded language and a syntax definition for the combination of the two languages. A generated parser then uses this combined syntax definition to parse templates written in the extended language.

Combining two languages is achieved by creating a new SDF module, which imports both syntax definitions. But only importing does not provide embedding, because the host language and the embedded language should be strictly separated. This separation is achieved by renaming all non-terminals by prefixing them with the name of the language. This renaming is necessary to keep the two languages strictly separated because e.g. both can provide `Id` or `Stm` non-terminal. If these syntax definitions are just imported directly, then there will be only one `Id` or `Stm`

non-terminal. Productions using these non-terminals in one grammar will then refer to the productions in another language and vice versa. This confusion can way to unwanted behavior of the resulting parser.

The imported languages are strictly separated from each other since the productions of the host language do not refer to non-terminals of the embedded language, and vice versa. The embedding is achieved by adding new productions to the combined syntax definitions which will allow using the embedded language statements or expressions in the place of host language statements or expressions. These production rules just connect both languages at the desired locations. This embedding is completely user-definable in the SDF module.

The location of the connection between the languages is indicated by special constructors. For example, productions which allow the embedded language statement to be used as the host language statement is marked by the constructor `FromTL`. And vice versa productions which allow the host language statement to stand instead of the embedded language statement uses the constructor `ToTL`.

An important point of the combined syntax definition is avoiding an ambiguity. But SDF provides a large set of features described in *Subsection 3.2.7 Solving ambiguity* which allows solving ambiguous expressions.

An example of the combined syntax definition is shown in *Listing 6.1.* The host language is Java in which the meta-language ElLang is embedded. The language ElLang provides simple meta-statements which allow generating and manipulating with Java commands.

```
module ElLang-J

imports
  Java-15-Prefixed
  ElLang-Prefixed

exports
  sorts

context-free syntax

  %% connecting production
  JavaId         -> ElLangVarRef  {avoid, cons("IdToTL")}
  ElLangVarRef   -> JavaId        {avoid, cons("IdFromTl")}

  JavaStm        -> ElLangStm     {avoid, cons("ToTL")}
  ElLangStm      -> JavaStm       {avoid, cons("FromTL")}

  %% ...

  %% solving ambiguity
  "method"       -> JavaId {reject}
  "template"     -> JavaId {reject}

priorities
      <JavaStm-CF> -> <ElLangStm-CF>
```

```
>       <JavaId-CF> -> <ElLangVarRef-CF>
>       <ElLangStm-CF> -> <JavaStm-CF>
>       <ElLangVarRef-CF> -> <JavaId-CF>
```

Listing 6.1: A definition of a combined syntax for two languages

6.2 Proposed template language ElLang-J

6.2.1 ElLang

As we have written above the template language is a mixture of *ElLang* language and the target language. ElLang was developed like a meta-language which allows us to manipulate and modify target language statements. This meta-language provides simple if, foreach, set, import statements and several more specific statements (like extension points), which will be described later. ELLANG has no expressional power without embedding it into some language.

6.3 Template description

The template language called ELLANG-J consists of target language statements and ElLang meta-statements. The main part of the template is the `element` statement which introduces an element definition. It can be completed by the `extends` statement with a name of the template which should be extended. The `element` statement includes interface definitions enclosed in the command `interface` followed by an interface name. It can also contain target language statements like variable declarations or static initializer which will create a part of a generated class. The interface definition can contain methods of the target language or a special statement `method template`. The method template includes target language statements as well as ELLANG statements. The method template will be applied on all methods of the specified interface.

```
package ${package};

import org...runtime.*;

element {
    public ${classname} { // constructor
    }

    implements interface ElementLocalClient {
        public void bindElPort(String portName, Object target) {
            /* ... */
        }
    }

    implements interface ${ports.port(name=line).signature} {
```

```
        method template {
            /* ... */
        }
    }
}
```

Listing 6.2: An example of a simple template

6.3.1 Meta variables

Meta variables are enclosed in `${name}`, where `name` is the name of a meta-variable. There are two types of meta-variables. The first type are queries (e.g. `${element.name}`) which we have introduced in *Subsection 5.1.4 Query module* and the second type are ElLang variables created by `set` statement. These variables can have a scalar form (e.g. `${a}`) or they can be written as arrays (e.g. `${b[1]}`). Another meta-variable or a literal (integer, string) can stand like an index of an array. Meta-variables can take the value of a meta-expression or a part of a XML element descriptor in case of the queries.

6.3.2 Meta expressions

Meta expressions support only `+` and `==` operators. As an operand literals (integer, string) as well as another meta variables can be used. Explicit quotation of meta-variables with `${...}` is not needed inside an expression.

```
$set( a = ports.port#count + 1)$
$set( b = ports.port(name=call).signature + ".java")$
```

Listing 6.3: An example of meta-expressions

6.3.3 Basic meta statements

ElLang offers several simple meta-statements which are useful for manipulating with target code.

6.3.3.1 Foreach cycle

Foreach cycle has the form:

```
"$foreach(" Id "in" VarRef ")$" Stm* "$end$" -> Stm {cons("Foreach")}
```

where `Id` is a name for a cycle meta-variable and `VarRef` is a meta-variable. The statements inside the foreach cycle are repeated as many times as is the number of

values stored in `VarRef`. The foreach cycle meta-variable with the name represented by `Id` obtains one value from the meta-variable `VarRef` in each loop.

```
$foreach(REMOTE_PORT in ${ports.port(type=REMOTE)})$
  boundedToRemoteRef[${i}] = null;
  $set ref[REMOTE_PORT.name] = i$
  $set i = i + 1$
$end$
```

Listing 6.4: An example of the foreach statement

6.3.3.2 Recursive foreach cycle

A recursive foreach cycle is an analogy of the foreach cycle shown above. It should be used in situations when we need to generate special target language constructs and normal foreach cycle cannot be used because of grammar restrictions. Recursive foreach cycle is defined:

```
"$rforeach(" Id "in" VarRef ")$" RecStm* "$final$" Stm* "$end$" -> Stm {cons("RForeach")}
```

where `Id`, `VarRef` has the same meaning like above. In the contrast to the normal foreach cycle, this recursive variant has two sections of statements separated by keyword `$final$` - the first one contains loop statements and the second one is a final section. Statements from this section will finish the foreach statement. The first section `RecStm*` is a list of statements which should contains special meta-statement `$recpoint$`. It has a special importance because it defines a location of the recursion.

The recursive foreach works as follows - if `VarRef` has no value (e.g. it is empty list of XML elements) then only statements `Stm*` from the final section are generated. Otherwise meta-variable with the name `Id` is set to the first value of the meta-variable `VarRef` and statements `RecStm*` are generated and all meta-variables with the name `Id` are evaluated. If meta-variable `VarRef` contains more values then the first is taken and it creates a new value of the meta-variable with the name `Id`. Then `$recpoint$` statement is found inside previous generated statements and its location is substituted with statements `RecStm*`. If the meta-variable `VarRef` contains no more values then `$recpoint$` statement is substituted with statements `Stm*` from the final section.

```
$rforeach(PORT in ${ports.port(type=REMOTE)})$
 if ("${PORT.name}".equals(portName)) {
     /* ... */
 } else $recpoint$
 $final$
  throw new ElementLinkException("Invalid port '"+portName+"'.");
$end$
```

Listing 6.5: An example of the recursive foreach statement

6.3.3.3 Condition statement

A condition statement is used for making branches in a code execution. The form of the if statement is simple:

```
"$if(" Expr ")$" Stm* "$end$" -> Stm  {prefer, cons("IF")}
"$if(" Expr ")$" Stm* "$else$" Stm* "$end$" -> Stm {cons ("IF")}
```

The main branch of if statement is generate only if `Expr` is not equal to `0` otherwise the *else* branch (if exists) is generated.

```
$if (BINDING.from.element.name != "this") $
 $if (BINDING.to.element.name != "this") $
  ((ElementLocalClient) subElements[${el[BINDING.to.element.name]}])
   .bindElPort("${BINDING.to.port}",
               ((ElementLocalServer) subElements[${el[BINDING.from.element.name]}])
                                                  .lookupElPort("${BINDING.from.port}");
 $end$
$end$
```

Listing 6.6: An example of the if statement

6.3.3.4 Set statement

Set statement serves for creating meta variables and changing their values.

```
"$set" VarRefPart "=" Expr "$" -> Stm {cons("Set")}
```

The scope of created variable is not restricted and it is accessible anywhere in a template code.

```
$set i = 0$
$foreach(ELEMENT in ${elements.element})$
  subElements[${i}] = new ${ELEMENT.class}();
  /* remember ELEMENT index in array */
  $set el[ELEMENT.name] = i$
  $set i = i + 1$
$end$
```

Listing 6.7: An example of the set statement

6.3.3.5 Import statement

The import statement can be used for importing parts of template code. Interface definitions and template methods are allowed to be imported.

```
$import("tests/simple/template_part_method.ellang")$
```

Listing 6.8: An example of import statement

6.3.4 Template hierarchy

Templates for elements can create a hierarchy structure. Each template can extend another template but cycles are not allowed. This hierarchy structure permits simple inheritance where ascendant element definition is extended by a descendant element definition. The solving of method collisions depends on a template designer.

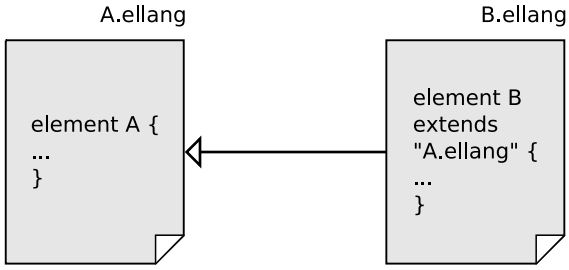


Figure 6.1: An example of template hierarchy

6.3.5 Template extension points

The proposed template language provides a concept of extension points. An extension point is definable location inside the code of the template which can be modified in a descendant templates. A simple extension point is defined by the meta-statement `$extPoint(name)$` but it can also bound several statements `$extPoint(name)$ ... $end$`.

In the descendant template a defined extension point can be modified by the statement `$defExtPoint(name)$ ... $end$`. The statements inside the definition substitute extension point with the name `name` as well as all statements bounded by that extension point.

6.3.6 Method templates

Selected types of connector elements just need to modify or register a call between two ports. An example of such situation is a simple log element. It has two ports - provided `in` and required `out`. It just generates a log entry when the call on the port `in` is emitted and then delegates the call to a port bounded to `out` port.

The method template is a part of an interface definition. The template is applied to all methods of the interface. It begins with reserved words `method template` followed by a Java block. Inside this statement special meta-variables with a name starting `method.` can be used. They refer to properties of a general method like a name, a return value, method parameters, ... During evaluation of the template they are filled with concrete values.

- `${method.name}` - the name of the method.

- `${method.returnVar}` - the name of the return variable if the method has a return type.
- `${method.declareReturnValue}` - declare a new return variable if the method has a return type.
- `${method.returnStm}` - generates a return statement if the method has a return type.
- `${method.variables}` - contains list of method parameters.

The example *Listing 6.9 Method templates* shows a typical use case of the method template, which adapts method call to a new interface.

```
implements interface ports.port(name=in) {
  method template {
    // declare return variable if it is needed
    ${method.declareReturnValue}
    // declare new variables which will participate in a method call
    String preVar;
    String postVar;

    /* fill preVar, postVar variable in according to a state of the element */

    System.out.println("The method \"${method.name}\" was called.");

    $if (method.returnVar) $
      ${method.returnVar} = this.target.${method.name}(preVar, ${method.variables}, postVar);
    $else$
      this.target.${method.name}(preVar, ${method.variables}, postVar);
    $end$

    //generates return statemene if it is needed
    ${method.returnStm}
  }
}
```

Listing 6.9: An example of method template declaration

6.4 Template evaluation

The template evaluation process consists of several steps assembled into a template evaluation module (the structure of the module is shown by *Figure 6.2*). Each step is implemented by a STRATEGO module providing a transformation of a selected part of the template or an assimilation of one or more meta-statements.

Figure 6.2 is showing individual tasks of the template evaluation. The first step is processing of the extends command which is done by the strategy `process-extends`. The program looks for the template, which name stands after `extends` command. The name of the extended template is considered as a path on the file system; thus it

can be absolute or relative to the actual directory. Founded template is parsed and its body is appended to the current template body. The evaluation process checks cyclic dependencies and if it finds a cycle the evaluation fails. But it does not care after overlapping parts (e.g. method with the same signature) of templates bodies. It lets a solution of this situation on a template developer.

The output is utilized by the strategy `process-import`. It expands the `import` meta-statement into the content of an imported file. The file is parsed by the strategy `parse-ElLang-J`, which uses the standard `sglri` tool and ElLang-J grammar definition stored in the file `ElLang-J.tbl` (the file should be accessible by the connector element generator). The imported file can contain only an interface definition, a Java method or a method template definition. This behavior is achieved by exporting multiple starting symbols in the grammar definition of ElLang-J language.

Now collecting information about the template is finished and it has to be prepared for the process of meta-statements evaluation. Preparation of the template is based on rewriting meta-statements to a unified form. For example, the meta-statement `if` can have to forms of notation. One is a simple `if` without else branch represented by the term `IF(condition, statement*)` and another one is `if-else` which has the term notation `IF(condition, statement*, else-statement*)`. If we permit both term representations of `if` meta-statement we have to write two different evaluation rules. Hence rewriting the meta-statements into an unified form is advantageous. In the case of the `if` example the unified form of the term `IF(condition, statement*)` is `IF(condition, statement*, [])`. The strategy `prepare-template` looks after a process of terms unification and it unifies the term representation of meta-statements `if`, `set` and `extPoint`.

After this stage the strategy `process-defextpoint` is called. It finds all `$extPoint(name)$` statements and substitutes them with statements declared in the corresponding `$defExtPoint(name)$`

After this step the template is in a normal form and assimilation of meta-variables and meta-statements can start. The first task is the evaluation of queries which refer to the input descriptor of a connector element. This process is implemented by the strategy `query` described in *Subsection 5.1.4 Query module.*

Next stage provides assimilation of meta-statements. This process is little bit more complex because it has to also include evaluation of meta-variables created by the `set` statement:

```
// main evaluation strategy
evaluation = /* ... */
       ; alltd(eval-stats)
       ; /* ...*/
// evaluate referencies to the input element descriptor
eval-varref = IdFromTL(innermost(EvalVarRef))
// evaluate meta-statements and meta-variables
eval-stats = eval-varref <+ eval-stat
```

```
// evaluate meta-statements
eval-stat = process-foreach <+ process-rforeach <+ process-if <+ process-set
```

The strategy `eval-stats` processes all meta-statements with help of the traversal strategy `alltd`. Using of the traversal strategy evokes evaluation of statements in the order which they were declared. The order is important for evaluation because the strategy has to figure out values of meta-variables and then propagates these values to following meta-statements. On the other hand some meta-statements can contain expressions. The evaluation strategy of these statements has to evaluate all meta-variables included in the expression, then it has to simplify the expression (evaluating of operators) and figures out its value. For this operation the best-fit strategy is *bottom-up* traversal. For example, the `foreach` meta-statement has to in each loop generates its body and evaluates it in according to the actual value of the cycle variable:

```
process-foreach =    ?Foreach(FVAR, INVAR, BODY);
             if <?[_|_]> INVAR    then
               <map(unfold-foreach(|FVAR, BODY))> INVAR
             else if <?[]> INVAR then
               !Empty // generates Empty statement
               else
                 <map(unfold-foreach(|FVAR, BODY))> [INVAR]
               end
             end
             ; alltd(eval-stats)

unfold-foreach(|FVAR, BODY) =    /* ... */
               ; !BODY
               ; bottomup(try(evaluate-varref(|variableXmlValue))) // evaluation of expr.
               ; /* ... */
```

The `eval-stats` combines all these operations as well as it includes assimilation of basic meta-statements (if, foreach, rforeach, set).

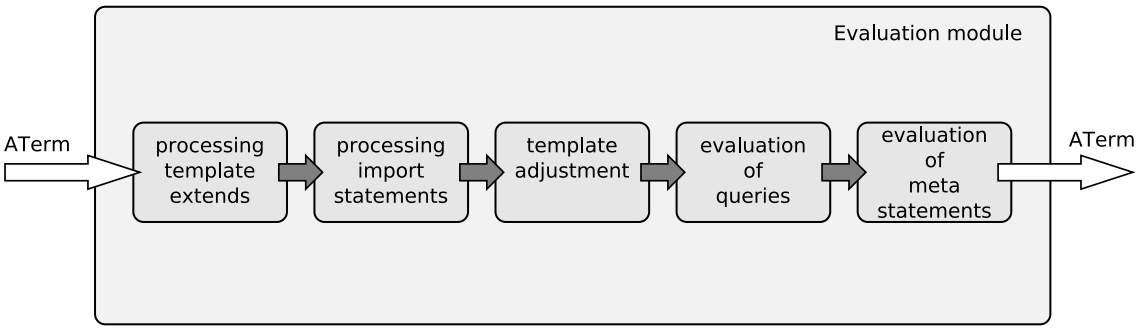


Figure 6.2: Template evaluation module

Chapter 7

Evaluation

7.1 Eligibility of StrategoXT

One of main objectives of this work is testing suitability of Stratego/XT toolkit for generation of software connectors. From a high level view Stratego/XT is a term rewriting system which parses an input language into terms, then runs transformations over terms and finally translates terms into an output language. Each stage of this process operates over a set of terms, which are the main representation of data in Stratego/XT. They hold a tree structure of an input program and allow us to modify parts of tree simply. This concept has been advantageous for our purposes because it allows separating implementation into relatively independent parts. But this separation is not absolute - the part, which is common for all three stages, is SDF (syntax definition formalism - see *Section 3.2 Syntax Definition Formalism SDF*) used for defining a grammar of an input language. The sharing of the grammar definition brings easy way of propagating changes in the grammar into the others modules of the code generator.

From our point of view usage of the Stratego/XT system have seemed as the best choice for transforming a low-level connector configuration into target code.

The parsing capability of Stratego/XT, which is based on SDF (syntax definition formalism - see *Section 3.2 Syntax Definition Formalism SDF*), fully satisfies our expectations. SDF offers a simple and fast method of defining a grammar of an input language. The great advantage of SDF, which help us, is that it imposes no restriction on the grammar. It allows several derivations for one input string, but on the other hand it offers a simple way of disambiguation, which makes writing language grammars really comfortable. Stratego/XT also supports automatic generation of parsers from a grammar described in SDF. The produced parser is an executable file, which takes like an input a file and produces terms. This automatic process is provided by Stratego/XT in the form of makefiles.

Terms representing a tree structure of the input sentence are playing the key

role in the next stage, where transformation strategies are applied on them. The strategies are described by the Stratego language, which provides a large set of commands modifying and transforming an abstract syntax tree represented by terms. The resulting Stratego program is compiled into executable form and can be used with a parser. Here we encountered a problem with long-time compilation of Stratego programs. The used compiler (Stratego/XT 0.16) does not use separated build units, but it compiles all Stratego code every time when the compilation is requested. This problem does not affect the code generator itself, because for defining a new connector element implementation, a developer just needs to write an element template (i.e. no code in the Stratego programming language). But it made development of Stratego/XT code generator slower and uncomfortable. The Stratego group plans to remove this disadvantage in a future version of Stratego/XT toolkit.

When all transformations are passed the final stage starts. It is called in order to rewrite terms into human readable format. This step is controlled by rules defined by *pretty print tables*. This definition is again derived from language grammar described in SDF. In this stage we reuse the existing pretty printer, which produces Java source code from terms. This reuse is typical for the Stratego/XT toolkit, because it includes a lot of already written programs (pretty printers for Java, C, SDF) and grammars definition (Java 1.4, Java 5, C, SDF, Stratego).

The prototype implementation does not use a great Stratego language feature called a *concrete syntax*. It allows using an input language while writing transformation strategies instead of using term notation. This behavior simplifies composing strategies and makes source code more readable, but it carries writing additional definitions of a *concrete syntax*. But in the context of our work we have not needed so powerful tool and we have rather focused on a transformation process itself.

The main disadvantage of the Stratego/XT system, which we have met during writing the prototype implementation of the generator, is its C implementation. All tools inside Stratego/XT package communicate via shell pipes, but when we need to connect them with a program implemented in Java, it carries implementation of a bridge (JNI call or executing an external process). This feature handicaps Stratego/XT from a view of Java developers.

The Stratego/XT engine is not also suitable for manipulation with system resources. Although, it implements a large set of system calls in the form of strategies (e.g. creating a file, writing to a file, reading directory content), usage of them is rigid and uncomfortable.

An indisputable advantage of Stratego/XT is that it is the active project without critical bugs, which has stable research and development background at Utrecht University in Netherlands.

7.2 Eligibility of ElLang-J

The language ElLang-J which has been proposed in this thesis, is a mixture of the Java programming language and the ElLang meta-language. It provides a comprehensive way of defining connector elements. It is used as a replacement of templates and generator classes in the previous solution [33]. The original template language was based on the Java language which was extended with meta-tags enclosed in % characters. These meta-tags were expanded by a specified generator class. It generates except simple names (like a class name, a package name) also bundles of source code. On the one hand this concept offers a way of evaluating arbitrary templates, but on the other hand it knows nothing about the grammar of a template language and it cannot check if the template is well written or not. And also dividing the template definition into a text template and its expanding class carries a lot of problems for connector element designers.

Therefore the newly designed template language tries to remove disadvantages of the existing generator and unifies a place, where a template is defined. The language itself provides a set of meta-statements, which can access an input connector element descriptor and control generation of output source code. These statements move the logic of template evaluation from a Java class into a template itself. Except common statements (*if*, *set*, *import*) the solution introduces a special meta-statements like *joint-points* or *recursive foreach* (see *Chapter 6 Template language*). These statements reflect special demands of element implementations, which were discovered during developing of the prototype. We have demonstrated the expressiveness of ElLang-J by rewriting all existing connector element definitions into a new format using ElLang-J. The comparison of the previous element definition and proposed template language is shown in appendix.

Chapter 8

Related work

The connector element generator presented by this thesis constitutes a comprehensive solution of connector generation.

However, there are several existing approaches of connector generation. One representative named *OpenWings* was mentioned in *Chapter 1 Introduction and motivation*. Generation system *Unicon* [42] is another project concerned in generation of connectors. It is an architectural description language for creating connectors. From the UniCon point of view the connector does not correspond to a separate compilation unit, but rather to a system call, linker instructions or other type of glue code. Because a connector mediates a connection between components, it is described by a *protocol*. The protocol itself consists of a *connector type* and list of *roles* (i.e. points through which a connector communicates with a component). UniCon tries to identify known types of connectors in accordance with a communication style (e.g. data flow - pipe, procedure call, data sharing - blackboard, ...). The connector type is the main abstraction in UniCon, because it denotes connector roles and component players (i.e. communication points of a component) which can cooperate together. Each connector is realized by an *expert*, which contains knowledge required to build a connector. It includes C source code fragments, templates, build scripts and semantics rules, which are checked before connector construction. The concept of UniCon system is really close to the generator [33] used in this work. But in UniCon there is no way how to easily define a new type of the connector. A new type has to be provided as an external module of the Unicon framework.

Nevertheless all these systems provide a solution of the connector generation problem they do not cover a complexity of the problem and they are not actively supported. Therefore only building a connector generator from scratch with help of the experiences from existing systems appears as the right way.

Connector generation is a wide subject which is influenced by many technologies and ideas. Thus the following sections introduce topics which are partially related to our work.

8.1 Methods of program synthesis

The code generator implemented in STRATEGO takes like an input an abstract descriptor of a connector element. The description is transformed into source code with help of templates. The idea of a program synthesis from an abstract description is not new and it helps reduce development time, errors and maintenance.

One representative is *AutoBayes* [37] - a fully automatic high level generator system developed in NASA which produces data analysis programs from statistical models. The model specifies statistical properties, their conditions and probabilities. From this description, AutoBayes generates optimized and commented[1] C++ code which can be then used in Matlab or Octave programs. A generation process is guided by a general algorithm schemes which include program fragments with open slots and constraints checked against a statistical model during code generation. The open slots are filled in according to the input model and described schema constraints. The generator can be easily extended with new control schema without modifying the kernel of the system. AutoBayes code generation system is completely implemented in SWI Prolog.

Another program synthesis tool is *AutoFilter* [44] as well implemented by Robust Software Engineering group at NASA Ames. It deals with generation of programs that solves estimation problems (e.g. computing an approximate attitude or a velocity vector) with special methods called generally Kalman filters. Given high-level model of a problem described in a form of linear or differential equations is automatically transformed into C/C++ (or Modula II) code, which can compute estimations in according to the described model. The code generation is also schema based, where a schema is a generic representation of a well-known algorithm which solves an input problem. The algorithm is described by a simple template programming language. An advantage of the generation system is pluggable *support modules* that produce target code.

Both systems mentioned above represents an idea of program synthesis from a high level description, which is close to our system. They contain well-designed part of target code generation that supports pluggable modules. On the other hand, each of them implements a solution of a domain specific problem that cannot be adapted to general connector generation.

Another approach of synthesizing programs is reuse existing code fragments and build them together in accordance with a high-level description. The solution proposed by the thesis is based on this idea of program constructing. It adapts code templates for different elements and brings them together. Our solution is not generic and is close to connector element generation. However, the general applications of this principle exist:

[1] The comments are reused in correctness proving of generated code.

Feature oriented programming is a paradigm for applications synthesis, analysis and optimization. The main idea of FOP is to build programs (and also classes) incrementally by composing features. Features are shown as basic building blocks of applications (which are interested for stakeholders) and feature characteristics are used to distinguishing programs within a family of related programs. These families are called software product lines [27]. The model of a product line architecture has three basic ideas - (a) identifying the similar set of features in a family of applications, (b) implementing each feature in one or more ways and (c) defining specific applications of product lines by the set of features that it supports and their implementations. This concept permits the feature to refine others features.

There are several implementation of feature refinements - one of them is *AHEAD (Algebraic Hierarchical Equations for Application Design)* [26] which is based on step-wise refinement. This system considers the feature as the primary unit of software modularity. The feature is not only source code (e.g. class), but also another hierarchical program representation (e.g. makefile, documentation, UML model, ...). From AHEAD theoretical view features are algebra operators and programs can be created by composition of such operators.

The idea of composing features is related to building connector from elements. Each feature would be associated with some type of element (e.g. log element has logging feature) and the whole connector would be built as a product line of specified elements. But currently FOP is only academic concept which is not widely accepted.

Aspect oriented programming [39] is a method which helps programmers to separate concerns. The AOP is primary focused on cross-cutting concerns. These concerns appears across many modules in a program (typical example is logging which is tangled and scattered in code). This is the contrast [25] with FOP which separates program features which are composing the resulting application. With AOP programmers instead of implementing functionality into an object they write a point-cut which *weaves* the aspects into objects. Hence objects can contain the basic logic which are not tangled with uninteresting code.

AOP identifies well-defined points in program flow (e.g. method execution, method call, field get, ...), which are called *joint-points*. Execution at a joint-point can be modified by an *aspect*. The aspect encapsulates information which joint-points affects it and how. Thus it introduces *point-cut* - a set of join-points in which the given aspect is interested. And in addition the aspect has to specify what happens at the join-points. The piece of the code which is executed at the specified point-cut is called *advice*. This code can access variables which are visible at given point-cut (e.g. class variables).

The leading Java implementation of this paradigm is *AspectJ* [5]. It provides all features of AOP described above. Although AOP is currently widely used even in a commercial environment, it is not really usable for connector element generation

based on the concept of the existing generator [33]. This is because AOP just affects predefined parts of code, but we need to adapt and modify whole element template as a solid unit.

8.2 Transformation languages

The proposed generator transforms a template into a target programming language with help of the Stratego language. Transformations are implemented by term rewriting strategies, but it is not only sole approach, which can be chosen.

One favorite method of a transformation is using extensible stylesheet language transformation (XSLT [9]) It belongs among XML-based languages used for the transformation of XML documents. It is most often used for converting XML documents into web pages or pdf documents but it can be also used for code generation. An input XML document serves as an input template which is processed by a XSLT processor. The processor is driven by a XSLT stylesheet file which describes transformation over the input document (represented as XML tree). Because an input template is in a XML language we do not need to define a grammar of the template. The main disadvantage of this approach is not-well readable language for defining template transformations (because it is based on XML) and also the limited access to system resources for XSLT processor.

Special group of transformation languages is composed of template languages. They provide a set of meta-commands which control generation of output code (e.g. HTML, XML, RTF). These languages are often designed for a general purpose, but favorite target domain, where they are used, is generation of web pages. The most known are Velocity [19] or FreeMarker [11], which offer accessing Java variables from a simple template language. Next representative is JET project, which template language is Java based and is similar to proposed language in this thesis. Java Emitter Templates (JET) [6] is a part of *Eclipse Modelling Framework*. It realizes implementation of code generation from an abstract model. The syntax of *JET* templates is based on JSP (Java Server Pages). Hence the template provides Java based meta language to access Java variables. Variables can be passed into the template via a template expander class and they can be easy used inside template code. The template itself is translated into a Java class during the evaluation stage. The class has a method `generate` which produces a result string (containing evaluated template). The advantage of this approach is easy learning template language based on Java. But the main problem of the JET and generally of all template engines is the grammar of the template. The engines do not know the target language, so they cannot define a complete grammar of a template language (they can only define the grammar of a meta-language). This is the main difference between our solution and the general template engines, because the proposed template language

has the grammar based on the grammar of the target language (Java) and the meta-language (*ElLang*). Thus all syntax errors in a template can be found during parsing stage, which makes writing templates more comfortable and error proof.

8.3 Term rewriting systems and grammar tools

The core of the proposed connector generator is a transformation system based on the STRATEGO language. STRATEGO is a member of a wide group composed of term-rewriting systems. It transforms terms with respect to a set of rewriting strategies. The closest systems to STRATEGO are the *ELAN* [7] system developed at LORIA centre, *ASF+SDF* [1] or the *Maude system* [15]. In comparison with Stratego, these tools are not so powerful and do not provide so wide set of supporting tools and grammar definitions. They are not also under so active research and development as STRATEGO.

For program transformations the grammar of an input language has to be specified. Chosen tool in this work was SDF (Syntax Definition Formalism) from STRATEGO/XT package, because it has several advantages in comparison with solutions based on *javacc* or on a combination of *lex* and *bison*. The grammar defined via SDF can be modularized into separate files, which allows re-use of parts of other grammar definitions. The defined grammar can be directly used for generation a language parser and further the same definition can be reused in the STRATEGO transformation language. The modularized structure of a grammar allows defining mixed grammars composed of already written grammars (e.g combination of meta-language *ElLang* and *Java* is resulting into *ElLang-J*). In contrast to *lex+bison*, which allow only class of LALR(1) grammars, SDF imposes no restrictions on the grammar. In SDF all derivations are produced and no implicit disambiguation will apply. Thus SDF provides a set of features (see *Subsection 3.2.7 Solving ambiguity*) which helps solving ambiguities and makes writing grammar definitions comfortable.

Chapter 9

Conclusion and future work

9.1 Summary of work

The main goal of this master thesis was an improvement of the existing connector generator ([33]) with the Stratego language. We have focused on a definition of a connector element implementation and we attempted to make writing the definition easier and more user friendly. We have presented an implementation of a connector generator, which offers templates for defining a connector element implementation. The template definition is based on a domain specific language ElLang-J which is a mixture of the Java language and our meta-language ElLang.

The ElLang is the core language which give us the way to access an input connector element description stored in XML format. It also contains meta-statements which manipulate with a target code and directly affect generated code. The language also allows simple inheritance among connector elements where child element inherits all code of the parent element implementation and it can modify specified parts of code. Another method how to modify parent element code is the principle of joint-points. The parent connector element implementation specifies such joint-points in the program execution. Then the child element can add new target code at the specified joint-points. These features give us a way of code reuse and they simplify an element implementation.

With this language we achieve separation of a connector element implementation from the connector generator.

The inner structure of the element transformer was designed in a way which allows us adding a new target language by specifying its mixture grammar and writing several Stratego language statements transforming an intermediate source code format into target code.

The main objective of the thesis was also testing an eligibility of Stratego/XT toolkit for generating target code of a connector element and making

transformation above it. STRATEGO/XT seems to be suitable for this objectives, because it provides simple a way to parse, transform and generate source code. It allows defining domain specific languages and their parsers. Transforming itself is based on strategies which allow step-by-step code modification.

As the problematic part of using the STRATEGO/XT toolkit we have detected its C-based implementation. STRATEGO/XT generates parsers and transformers in C code and compile it into an executable format. This carries demand for their recompilation on different target platforms and includes an overload due to necessity to over bridge Java and C implementations.

9.2 Prototype implementation

This thesis is supplemented by a prototype implementation of the connector element generator based on the STRATEGO/XT package. The solution extends already implemented connector generator [33]. Our generator also re-implements all connector element templates from the previous generation system. Thus it is at least powerful as the existing one.

We have included the described connector element generator to the latest version of the existing generator. The actual version of this implementation can be found at `http://aiya.ms.mff.cuni.cz/~bures/congen/websvn/`.

9.3 Future work

One of the future tasks is making the generation of connector elements faster. Current solution generates Java code, which is subsequently compiled by a Java language compiler (*javac*). The slowest part of the connector generation process is the compilation of Java code. Hence it seems advantageous to precompile templates into Java bytecode and then realize transformations of bytecode. With this approach we will be able to omit the Java code compilation stage and speed up generation of connectors.

The part which also needs improvement is the STRATEGO compiler. As we have mentioned in *Section 7.1 Eligibility of StrategoXT* the compiler of the STRATEGO language does not use separate compilation units and it compiles whole code every time.

The thesis has proposed the template language ELLANG-J, which is a mixture of Java and the ELLANG language. It is transformed into Java code, but in situations when we want to connect e.g. C++ components we would like to prefer generating C++ code. Thus implementing ELLANG-C++ or ELLANG-C# clone of ELLANG-J could be an interesting future task.

Bibliography

[1] Centrum voor Wiskunde en Informatica, AFS+SDF Meta-environement, http://www.asfsdf.org/.

[2] ArchJava, http://www.archjava.org/.

[3] Microsoft Corporation, Component Object Model, http://www.microsoft.com/com/.

[4] Object Management Group, CORBA component model, http://www.omg.org/technology/documents/formal/components.htm.

[5] Eclipse AspectJ, http://www.aspectj.org/.

[6] Eclipse JET (Java Emitter Templates), http://www.eclipse.org/emft/projects/jet/.

[7] LORIA, Laboratoire Lorrain de Recherche en Informatique et ses Applications, ELAN, http://elan.loria.fr/.

[8] Sun Microsystems, Enterprise JavaBeans, http://java.sun.com/ejb/.

[9] The extensible stylesheet language transformation, http://www.w3.org/TR/xslt.

[10] Fractal component model, http://fractal.objectweb.org/.

[11] FreeMarker, http://www.freemarker.org/.

[12] Intrinsyc Software, J-Integra, http://j-integra.intrinsyc.com/.

[13] Sun Microsystems, Java Native Interface specification, http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html.

[14] JNBridge, http://www.jnbridge.com/.

[15] Department of Computer Science, University of Illinois at Urbana-Champaign, The Maude System, http://maude.cs.uiuc.edu/.

[16] General Dynamics C4 Systems, Openwings, `http://www.openwings.org/`.

[17] SOFA component system, `http://sofa.objectweb.org/`.

[18] Stratego/XT, `http://www.stratego-language.org/`.

[19] The Apache Jakarta Project, Velocity, `http://jakarta.apache.org/velocity/`.

[20] Sun Microsystems, Inc., Java Remote Method Specification, `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.htm`, 2001.

[21] Sun Microsystems, Inc., Java Message Service Specification, `http://java.sun.com/products/jms/docs.html`, 2002.

[22] Sun Microsystems, Inc., JavaSpaces Service Specification, `http://www.sun.com/software/jini/specs/jini1.2html/js-title.html`, 2002.

[23] Object Management Group, Common Object Request Broker Architecture: Core Specification, `http://www.omg.org/docs/formal/04-03-12.pdf`, 2004.

[24] Object Management Group, Deployment and Configuration of Component Based Distributed Applications Specification, `http://www.omg.org/docs/ptc/04-08-02.pdf`, 2004.

[25] Apel, S., Leich, T., and Saake, G. Aspectual mixin layers: Aspects and features in concert, 2006, `citeseer.ist.psu.edu/apel06aspectual.html`.

[26] Batory, D. Feature-oriented programming and the ahead tool suite, `citeseer.ist.psu.edu/batory04featureoriented.html`.

[27] Batory, D. S. Product-line architectures, aspects, and reuse (tutorial session). In *International Conference on Software Engineering* (2000), p. 832.

[28] Bravenboer, M., de Groot, R., and Visser, E. Metaborg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In *Participants Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)* (Braga, Portugal, July 2005).

[29] Bravenboer, M., and Visser, E. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)* (Vancouver, Canada, October 2004), D. C. Schmidt, Ed., ACM Press, pp. 365–383.

[30] Bulej, L., and Bures, T. A connector model suitable for automatic generation of connectors. Tech. Rep. 2003/1, Dep. of SW Engineering, Charles University, Prague, January 2003.

[31] Bulej, L., and Bures, T. Using connectors for deployment of heterogeneous applications in the context of omg d&c specification, 2005, `http://citeseer.ist.psu.edu/bulej05using.html`.

[32] Bures, T. Automated synthesis of connectors for heterogeneous deployment, January 2005, `http://citeseer.ist.psu.edu/bures05automated.html`.

[33] Bures, T. *Generating Connectors for Homogeneous and Heterogeneous Deployment*. PhD thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, 2006.

[34] E. Dolstra, E. V. Building interpreters with rewriting strategies. Tech. Rep. UU-CS-2002-022, Institute of Information and Computing Sciences, Utrecht University, 2002.

[35] E. Visser, E. Scannerless generalized-LR parsing. Tech. Rep. P9707, Programming Research Group, University of Amsterdam, July 1997.

[36] E.Visser. Program transformation course, 2005, `http://www.cs.uu.nl/wiki/Pt`.

[37] Fischer, B., and Schumann, J. Autobayes: A system for synthesizing data analysis programs, `citeseer.ist.psu.edu/fischer00autobayes.html`.

[38] Galik, O., and Bures, T. Generating connectors for heterogeneous deployment. In *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware* (New York, NY, USA, 2005), ACM Press, pp. 54–61.

[39] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds., vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.

[40] M. Bravenboer, A. van Dam, K. O., and Visser, E. Program transformation with scoped dynamic rewrite rules. Tech. Rep. UU-CS-2005-005, Institute of Information and Computing Sciences, Utrecht University, 2005.

[41] Mehta, N. R., Medvidovic, N., and Phadke, S. Towards a taxonomy of software connectors. In *Proceedings of the 22nd International Conference on Software Engineering* (2000).

[42] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. Abstractions for software architecture and tools to support them. *Software Engineering 21*, 4 (1995), 314–335.

[43] Visser, E. Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering (GPCE'02)* (Pittsburgh, PA, USA, October 2002), D. Batory, C. Consel, and W. Taha, Eds., vol. 2487 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 299–315.

[44] Whittle, J., and Schumann, J. Automating the implementation of Kalman filter algorithms. *ACM Trans. Math. Softw. 30*, 4 (2004), 434–453.

Appendix A

Examples of source code

A.1 Template from previous connector generator

The previous version of the connector generator [33] divides a definition of connector implementation into two parts - a text file, which represents a static part of a template, and a Java class representing a dynamic part. This example shows a static part of a default template for a composite element. In comparison with a template presented in *Section A.3* it seems much more simple but in fact its logic is hidden into a Java class (see *Listing A.4 Generated code*), which expands tags quoted in **%** and generates the rest of code.

```
package %PACKAGE%;

public class %CLASS% implements
   org.objectweb.dsrg.connector.ElementLocalServer,
   org.objectweb.dsrg.connector.ElementLocalClient,
   org.objectweb.dsrg.connector.ElementRemoteServer,
   org.objectweb.dsrg.connector.ElementRemoteClient {

  protected org.objectweb.dsrg.connector.Element[] subElements;

  protected org.objectweb.dsrg.connector.RemoteRefBundle[] boundedToRemoteRef;

  public %CLASS%() {
  }

%INIT_METHODS%

}
```

As we have mentioned above the previous version of the generator uses a Java class for evaluation of a text template:

```
/**
 * Source code generator for composite elements.
 */
public class CompositeGenerator extends BaseGenerator {

    /**
     * Runtime package prefix.
     */
    private static final String RUNTIME_PACKAGE =
        "org.objectweb.dsrg.connector";

    /**
     * Map capturing the assignment of element instances to indices in the array
```

```
     * of element instances.
     */
    protected Map<ResolvedElementInstance, String> elementInstMap;

    /**
     * Map capturing the assignment of remote port names to indices in the array
     * of remote port target references.
     */
    protected Map<String, String> remotePortMap;


    /* ************************************************************************
     * CONSTRUCTORS
     * ************************************************************************/

    /**
     * Creates a new <code>CompositeGenerator</code> instance given an
     * <code>ElementGenerator</code>.
     *
     * @param elemgen
     */
    public CompositeGenerator (ElementGenerator elemgen) {
        super (elemgen);

        elementInstMap = new HashMap<ResolvedElementInstance, String> ();
        remotePortMap = new HashMap<String, String> ();
    }


    /* ************************************************************************
     * JImplGeneratorInterface
     * ************************************************************************/

    /**
     * Initializes the element instance and remote port maps and calls the
     * <code>generate()</code> method of the superclass.
     */
    public synchronized void generate (
        throws ActionException {

        /*
         * Create a map of element instances for which the element instance name
         * is the key and the index in the array of element instances is the
         * value.
         */
        elementInstMap.clear ();

        int instanceNo = 0;
        for (ResolvedElementInstance instance : ad.rei.subElementInst) {
            elementInstMap.put (instance, Integer.toString (instanceNo));
            instanceNo++;
        }

        /*
         * Create a map of remote ports for which the remote port name is the
         * key and the index in the array of remote port target references is
         * the value.
         */
        remotePortMap.clear ();

        int portNo = 0;
        for (ResolvedElementPort port : ad.rei.port) {
            if (port.port.type == PortType.REMOTE) {
                remotePortMap.put (port.port.portName, Integer.toString (portNo));
                portNo++;
            }
        }

        /*
         * Finally, propagate the call to the superclass.
         */
        super.generate (ed, ad, parameters, pd);
    }


    /* ************************************************************************
     * ContentProviderInterface
     * ************************************************************************/

    /**
     * Provides content for generation of Java implementation of a composite
     * connector element. Recognizes the following content identifiers:
     *
     * <dl>
```

```
 *   <dt>COMPOSITEL_METHODS</dd>
 *     Provides Java code of composite element implemenation
 *     methods: <code>initializeArchitecture()</code>.
 *   <dt>ELEMENT_METHODS</dd>
 *     Provides Java code of Element interface methods:
 *     <code>getElDescription()</code>.
 *   <dt>LOCALSERVER_METHODS</dd>
 *     Provides Java implementation of ElementLocalServer interface
 *     methods: <code>lookupElPort()</code>.
 *   <dt>LOCALCLIENT_METHODS</dd>
 *     Provides Java implementation of ElementLocalClient interface
 *     methods: <code>bindElPort()</code>, and <code>unbindElPort()</code>.
 *   <dt>REMOTESERVER_METHODS</dd>
 *     Provides Java implementation of ElementRemoteServer interface
 *     methods: <code>lookupElRemotePort()</code>,
 *     <code>listElRemotePorts()</code>.
 *   <dt>REMOTECLIENT_METHODS</dd>
 *     Provides Java implementation of ElementRemoteClient interface
 *     methods: <code>bindElRemotePort()</code>,
 *     <code>unbindElRemotePort()</code>, and
 *     <code>getElRemoteTarget()</code>.
 * </dl>
 */
public String getContent (String identifier)
    throws TemplateProcessingException {

    StringBuilder output = new StringBuilder ();

    if ("COMPOSITE_METHODS".equals (identifier)) {
        implementInitializeArchitecture (output);
    } else if ("ELEMENT_METHODS".equals (identifier)) {
        implementGetElDescription (output);
        // implementSetElReconfigurationHandler (output);
    } else if ("LOCALSERVER_METHODS".equals (identifier)) {
        implementLookupElPort (output);
    } else if ("LOCALCLIENT_METHODS".equals (identifier)) {
        implementBindElPort (output);
        implementUnbindElPort (output);
    } else if ("REMOTESERVER_METHODS".equals (identifier)) {
        implementLookupElRemotePort (output);
        implementListElRemotePorts (output);
    } else if ("REMOTECLIENT_METHODS".equals (identifier)) {
        implementBindElRemotePort (output);
        implementUnbindElRemotePort (output);
        implementGetElRemoteTarget (output);
    } else if ("RECONFIGURATION_METHODS".equals (identifier)) {
        implementInvalidateElPort (output);
        // implementInvalidateElRemotePort (output);
    } else {
        return super.getContent (identifier);
    }

    return output.toString ();
}


/**************************************************************************
 * Composite Method Generator
 **************************************************************************/

/**
 * Provides Java code of the element initialization method:
 * <code>initializeArchitecture()</code>.
 *
 * @throws TemplateProcessingException
 */
protected void implementInitializeArchitecture (StringBuilder output)
    throws TemplateProcessingException {

    Formatter result = new Formatter (output);

    /*
     * Method signature.
     */
    result.format (
        "\t"+"private void initializeArchitecture ()\n"+
        "\t\t"+"throws %s.ElementLinkException {\n\n",
        RUNTIME_PACKAGE
    );

    /*
     * Initialize element instance and remote port reference arrays.
     */
    result.format (
```

```
    "\t\t"+"subElements = new %s.Element[%d];\n",
    RUNTIME_PACKAGE, elementInstMap.size ()
);
result.format (
    "\t\t"+"remoteTargetRefs = new %s.RemoteRefBundle[%d];\n\n",
    RUNTIME_PACKAGE, remotePortMap.size ()
);


/*
 * Instantiate subelements.
 */
result.format (
    "\t\t"+"// instantiate subelements\n"+
    "\t\t"+"try {\n"
);

for (ResolvedElementInstance inst : ad.rei.subElementInst) {
    /*
     * Generate the subelement implementation and get the package
     * which contains the implementation. Add the newly generated
     * package into run-time dependencies of this element.
     */
    PackageDescriptor instPd;

    try {
        instPd = elemgen.generate (new AdaptationDescriptor (ad, inst));

    } catch (ElementGeneratorException e) {
        throw new TemplateProcessingException (e);
    }

    pd.runDependsOnPackage.add (instPd.packageName);

    /*
     * Instantiate subelements and provide them with
     * ReconfigurationHandler interface.
     */
    String implClass = Property.findFirst (instPd.execParam, "implClass");
    JavaNames implClassJN = new JavaNames (cam, instPd.packageName, implClass);
    result.format (
        "\t\t\t"+"// inst %s, type %s, impl %s\n"+
        "\t\t\t"+"subElements[%s] = new %s (parentUnit, false);\n",
        inst.instanceName, inst.elementType.typeName, inst.elementImplName,
        elementInstMap.get (inst), implClassJN.absoluteClassName
    );
}

result.format (
    "\n"+
    "\t\t"+"} catch (Exception e) {\n"+
    "\t\t\t"+"throw new %s.ElementLinkException (e);\n"+
    "\t\t"+"}\n\n",
    RUNTIME_PACKAGE
);


/*
 * Provide server subelements with reconfiguration handler.
 */
result.format (
    "\t\t"+"// distribute reconfiguration handler to server subelements\n"+
    "\t\t"+"distributeReconfigurationHandler ();\n"
);


/*
 * Bind subelements together. Only do this for real bindings, i.e.
 * bindings where both elements are set and the ports have complementary
 * types. The other bindings serve for delegation and subsumption of
 * this element ports to subelement ports.
 */
for (ResolvedElementBinding bind : ad.rei.binding) {
    // result.format ("\n");

    if (bind.isBinding ()) {
        /*
         * Both elements are set, just get the correct order for
         * formatting.
         */
        ResolvedElementInstance serverElement = bind.getServerElement ();
        String serverPortName = bind.getServerPortName ();
        String serverElementIndex = elementInstMap.get (serverElement);
```

```
            ResolvedElementInstance clientElement = bind.getClientElement ();
            String clientPortName = bind.getClientPortName ();
            String clientElementIndex = elementInstMap.get (clientElement);

            result.format (
                "\n"+
                "\t\t"+"// bind %s.%s -> %s.%s\n"+
                "\t\t"+"((%s.ElementLocalClient) subElements[%s]).bindElPort (\"%s\",\n"+
                "\t\t\t"+ "((%s.ElementLocalServer) subElements[%s]).lookupElPort (\"%s\"));\n",
                clientElement.instanceName, clientPortName,
                serverElement.instanceName, serverPortName,
                RUNTIME_PACKAGE, clientElementIndex, clientPortName,
                RUNTIME_PACKAGE, serverElementIndex, serverPortName
            );
        }
    }


    /*
     * Closing brace...
     */
    result.format ("\t}\n\n");
}


/*************************************************************************
 * Element Method Generator
 *************************************************************************/

/**
 * Provides Java code of the element description method:
 * <code>getElDescriptionInfo()</code>.
 *
 * @throws TemplateProcessingException
 */
protected void implementGetElDescription (StringBuilder output)
    throws TemplateProcessingException {

    Formatter result = new Formatter (output);

    /*
     * Method signature.
     */
    result.format ("\t"+"public String getElementInfo (String indent) {\n");

    /*
     * Construct the element information.
     */
    result.format (
        "\t\t"+"StringBuilder result = new StringBuilder ();\n\n"+
        "\t\t"+"result.append (indent +\"Implementation: \\\"%s\\\"\\n\");\n",
        ad.rei.elementImplName
    );

    for (ResolvedElementInstance inst : ad.rei.subElementInst) {
        result.format (
            "\t\t"+"result.append (indent +\"    Sub-element: \\\"%s\\\"\\n\");\n",
            inst.instanceName
        );

        result.format (
            "\t\t"+"result.append (subElements[%s].getElementInfo (indent +\"        \"));\n",
            elementInstMap.get (inst)
        );
    }

    result.format (
        "\n"+
        "\t\t"+"return result.toString ();\n"
    );

    /*
     * Closing brace...
     */
    result.format ("\t"+"}\n\n");
}


/*************************************************************************
 * ElementLocalServer Method Generators
 *************************************************************************/

protected void implementLookupElPort (StringBuilder output)
    throws TemplateProcessingException {
```

```
Formatter result = new Formatter (output);

/*
 * Method signature.
 */
result.format (
    "\t"+"public final Object lookupElPort (String portName)\n"+
    "\t\t"+"throws %s.ElementLinkException {\n\n",
    RUNTIME_PACKAGE
);

/*
 * Lookup a PROVIDED port of given name and return reference to it.
 */
result.format ("\t\t");
for (ResolvedElementPort providedPort : ad.rei.port) {
    if (providedPort.port.type == PortType.PROVIDED) {
        result.format (
            "if (\"%s\".equals (portName)) {\n",
            providedPort.port.portName
        );

        /*
         * Find the first binding describing the delegation of this
         * element provided port to a subelement provided port and
         * delegate the lookup method to the subelement instance. The
         * delegation binding has one of the elements set to null to
         * signify this parent element.
         */
        for (ResolvedElementBinding bind : ad.rei.binding) {
            if (bind.isDelegation (providedPort.port.portName)) {
                String childPortName =
                    bind.getChildPortName ();
                ResolvedElementInstance childElement =
                    bind.getChildElement ();
                String childElementIndex =
                    elementInstMap.get (childElement);

                /*
                 * Get the reference from the child element.
                 */
                result.format (
                    "\t\t\t"+"// inst %s, type %s, impl %s\n"+
                    "\t\t\t"+"Object result = ((%s.ElementLocalServer)\n"+
                    "\t\t\t\t"+"subElements [%s]).lookupElPort (\"%s\");\n\n",
                    childElement.instanceName, childElement.elementType.typeName,
                        childElement.elementImplName,
                    RUNTIME_PACKAGE,
                    childElementIndex, childPortName
                );

                /*
                 * Register the reference with the DCM if top-level
                 * and return the result.
                 */
                result.format (
                    "\t\t\t"+"if (isTopLevel) {\n"+
                    "\t\t\t\t"+"dcm.reregisterConnectorUnitReference (\n"+
                    "\t\t\t\t\t"+"parentUnit, portName, result);\n"+
                    "\t\t\t"+"}\n\n"+
                    "\t\t\t"+"return result;\n\n"
                );

                /*
                 * For local ports, the delegation must be 1:1, so we exit
                 * the loop when the first delegation binding has been
                 * found.
                 */
                break;
            }
        }

        result.format ("\t\t} else ");
    }
}

/*
 * Throw an exception if the port is unknown.
 */
result.format (
    "{\n"+
    "\t\t\t"+"throw new %s.ElementLinkException (\n"+
    "\t\t\t\t"+"\"Invalid provided port '\"+ portName +\"'.\");\n"+
```

```
        "\t\t"+"}\n"+
        "\t"+"}\n\n",
        RUNTIME_PACKAGE
    );
}


/*************************************************************************
 * ElementLocalClient Method Generators
 ************************************************************************/

protected void implementBindElPort (StringBuilder output)
    throws TemplateProcessingException {

    Formatter result = new Formatter (output);

    /*
     * Method signature.
     */
    result.format (
        "\t"+"public final void bindElPort( String portName, Object target)\n"+
        "\t\t"+"throws %s.ElementLinkException {\n\n",
        RUNTIME_PACKAGE
    );

    /*
     * Lookup a REQUIRED port of given name and bind it to the target
     * object.
     */
    result.format ("\t\t");
    for (ResolvedElementPort requiredPort : ad.rei.port) {
        if (requiredPort.port.type == PortType.REQUIRED) {
            result.format (
                "if (\"%s\".equals (portName)) {\n",
                requiredPort.port.portName
            );

            /*
             * Find the binding describing the subsumption of a subelement
             * required port to this element required port and delegate
             * the bind method to the subelement instance. The subsumption
             * binding has one of the elements set to null to signify "this"
             * parent element.
             */
            for (ResolvedElementBinding bind : ad.rei.binding) {
                if (bind.isSubsumption (requiredPort.port.portName)) {
                    String childPortName =
                        bind.getChildPortName ();
                    ResolvedElementInstance childElement =
                        bind.getChildElement ();
                    String childElementIndex =
                        elementInstMap.get (childElement);

                    result.format (
                        "\t\t\t"+"// inst %s, type %s, impl %s\n"+
                        "\t\t\t"+"((%s.ElementLocalClient)\n"+
                        "\t\t\t\t"+"subElements[%s]).bindElPort(\"%s\", target);\n",
                        childElement.instanceName, childElement.elementType.typeName,
                            childElement.elementImplName,
                        RUNTIME_PACKAGE,
                        childElementIndex, childPortName
                    );
                }
            }

            result.format (
                "\n"+
                "\t\t"+"} else ");
        }
    }

    /*
     * Throw an exception if the port is unknown.
     */
    result.format (
        "{\n"+
        "\t\t\t"+"throw new %s.ElementLinkException (\n"+
        "\t\t\t\t"+ "\"Invalid required port '\"+ portName +\"'.\");\n"+
        "\t\t"+"}\n"+
        "\t"+"}\n\n",
        RUNTIME_PACKAGE
    );
}
```

```
protected void implementUnbindElPort (StringBuilder output)
    throws TemplateProcessingException {

    Formatter result = new Formatter (output);

    /*
     * Method signature.
     */
    result.format (
        "\t"+"public final void unbindElPort (String portName)\n"+
        "\t\t"+"throws %s.ElementLinkException {\n\n",
        RUNTIME_PACKAGE
    );

    /*
     * Lookup a REQUIRED port of given name and unbind it
     */
    result.format ("\t\t");
    for (ResolvedElementPort requiredPort : ad.rei.port) {
        if (requiredPort.port.type == PortType.REQUIRED) {
            result.format (
                "if (\"%s\".equals (portName)) {\n",
                requiredPort.port.portName
            );

            /*
             * Find the binding describing the subsumption of a subelement
             * required port to this element required port and delegate
             * the unbind method to the subelement instance. The subsumption
             * binding has one of the elements set to null to signify "this"
             * parent element.
             */
            for (ResolvedElementBinding bind : ad.rei.binding) {
                if (bind.isSubsumption (requiredPort.port.portName)) {
                    String childPortName =
                        bind.getChildPortName ();
                    ResolvedElementInstance childElement =
                        bind.getChildElement ();
                    String childElementIndex =
                        elementInstMap.get (childElement);

                    result.format (
                        "\t\t\t"+"// inst %s, type %s, impl %s\n"+
                        "\t\t\t"+"((%s.ElementLocalClient)\n"+
                        "\t\t\t\t"+"subElements [%s]).unbindElPort (\"%s\");\n",
                        childElement.instanceName, childElement.elementType.typeName,
                            childElement.elementImplName,
                        RUNTIME_PACKAGE,
                        childElementIndex, childPortName
                    );
                }
            }

            result.format (
                "\n"+
                "\t\t"+"} else ");
        }
    }

    /*
     * Throw an exception if the port is unknown.
     */
    result.format (
        "{\n"+
        "\t\t\t"+"throw new %s.ElementLinkException (\n"+
        "\t\t\t\t"+"\"Invalid required port '\"+ portName +\"'.\");\n"+
        "\t\t"+"}\n"+
        "\t"+"}\n\n",
        RUNTIME_PACKAGE
    );
}


/*************************************************************************
 * ElementRemoteServer Method Generators
 *************************************************************************/

protected void implementLookupElRemotePort (StringBuilder output)
    throws TemplateProcessingException {

    Formatter result = new Formatter (output);

    /*
```

```
         * Method signature.
         */
        result.format (
            "\t"+"public final %s.RemoteRefBundle lookupElRemotePort (\n"+
            "\t\t"+"String portName)\n"+
            "\t\t"+"throws %s.ElementLinkException {\n\n",
            RUNTIME_PACKAGE, RUNTIME_PACKAGE
        );

        /*
         * Result value.
         */
        result.format (
            "\t\t"+"%s.RemoteRefBundle result =\n"+
            "\t\t\t"+"new %s.RemoteRefBundle ();\n\n",
            RUNTIME_PACKAGE, RUNTIME_PACKAGE
        );

        /*
         * Lookup a REMOTE port of given name and return a bundle of typed
         * references to it.
         */
        result.format ("\t\t");
        for (ResolvedElementPort serverPort : ad.rei.port) {
            if (serverPort.port.type == PortType.REMOTE) {
                result.format (
                    "if (\"%s\".equals (portName)) {\n",
                    serverPort.port.portName
                );

                /*
                 * Collect typed references to all subelement remote server
                 * ports this element remote server port delegates to.
                 */
                for (ResolvedElementBinding bind : ad.rei.binding) {
                    if (bind.isDelegation (serverPort.port.portName)) {
                        String childPortName =
                            bind.getChildPortName ();
                        ResolvedElementInstance childElement =
                            bind.getChildElement ();
                        String childElementIndex =
                            elementInstMap.get (childElement);

                        result.format (
                            "\n"+
                            "\t\t\t"+"// inst %s, type %s, impl %s\n"+
                            "\t\t\t"+"if (subElements[%s] instanceof %s.ElementRemoteServer) {\n"+
                            "\t\t\t\t"+"// lookup %s.%s\n"+
                            "\t\t\t\t"+"result.addRefBundle (\n"+
                            "\t\t\t\t\t"+"((%s.ElementRemoteServer)\n"+
                            "\t\t\t\t\t"+"subElements[%s]).lookupElRemotePort (\"%s\"));\n"+
                            "\t\t\t"+"}\n",
                            childElement.instanceName, childElement.elementType.typeName,
                                childElement.elementImplName,
                            childElementIndex, RUNTIME_PACKAGE,
                            childElement.instanceName, childPortName,
                            RUNTIME_PACKAGE,
                            childElementIndex, childPortName
                        );
                    }
                }

                result.format (
                    "\n"+
                    "\t\t"+"} else ");
            }
        }

        /*
         * Throw an exception if the port is unknown, otherwise return the
         * reference bundle.
         */
        result.format (
            "{\n"+
            "\t\t\t"+"throw new %s.ElementLinkException (\n"+
            "\t\t\t\t"+"\"Invalid remote server port '\"+portName+\"'.\");\n"+
            "\t\t"+"}\n\n"+
            "\t\t"+"return result;\n"+
            "\t"+"}\n\n",
            RUNTIME_PACKAGE
        );
    }
```

```
protected void implementListElRemotePorts (StringBuilder output)
    throws TemplateProcessingException {

    Formatter result = new Formatter (output);

    /*
     * Method signature.
     */
    result.format (
        "\t"+"public final String[] listElRemotePorts(){\n");

    /*
     * Create a String array with names of remote ports.
     */
    String prefix = "\t\t"+"return new String[]{\n"+"\t\t\t";
    for (ResolvedElementPort remotePort : ad.rei.port) {
        if (remotePort.port.type == PortType.REMOTE) {
            result.format (
                "%s"+"\"%s\"", prefix, remotePort.port.portName
            );

            prefix = ", ";
        }
    }

    /*
     * Close the array constructor and method declaration.
     */
    result.format (
        "\n"+
        "\t\t"+"};\n"+
        "\t"+"}\n\n"
    );
}


/*************************************************************************
 * ElementRemoteClient Method Generators
 *************************************************************************/

protected void implementBindElRemotePort (StringBuilder output)
    throws TemplateProcessingException {

    Formatter result = new Formatter (output);

    /*
     * Method signature.
     */
    result.format (
        "\t"+"public final void bindElRemotePort(\n"+
        "\t\t\t"+"String portName,\n"+
        "\t\t\t"+"%s.RemoteRefBundle refBundle)\n"+
        "\t\t"+"throws %s.ElementLinkException{\n\n",
        RUNTIME_PACKAGE, RUNTIME_PACKAGE
    );

    /*
     * Lookup a REMOTE client port of given name and bind it to the remote
     * target using a bundle of references.
     */
    result.format ("\t\t");
    for (ResolvedElementPort clientPort : ad.rei.port) {
        if (clientPort.port.type == PortType.REMOTE) {
            result.format (
                "if(\"%s\".equals(portName)){\n",
                clientPort.port.portName
            );

            /*
             * Push the remote reference bundle to all the elements that
             * subsume their remote client port to this element remote
             * client port.
             */
            for (ResolvedElementBinding bind : ad.rei.binding) {
                if (bind.isSubsumption (clientPort.port.portName)) {
                    String childPortName =
                        bind.getChildPortName ();
                    ResolvedElementInstance childElement =
                        bind.getChildElement ();
                    String childElementIndex =
                        elementInstMap.get (childElement);

                    result.format (
                        "\n"+
```

```
                        "\t\t\t"+"// inst.%s, type.%s, impl.%s\n"+
                        "\t\t\t"+"if (subElements[%s] instanceof %s.ElementRemoteClient) {\n"+
                        "\t\t\t\t"+"// bind.%s.%s\n"+
                        "\t\t\t\t"+"((%s.ElementRemoteClient)\n"+
                        "\t\t\t\t\t"+"subElements[%s]).bindElRemotePort(\"%s\", refBundle);\n"+
                        "\t\t\t"+"}\n",
                        childElement.instanceName, childElement.elementType.typeName,
                            childElement.elementImplName,
                        childElementIndex, RUNTIME_PACKAGE,
                        childElement.instanceName, childPortName,
                        RUNTIME_PACKAGE,
                        childElementIndex, childPortName
                    );
                }
            }

            /*
             * Keep the remote reference bundle for future queries.
             */
            result.format (
                "\n"+
                "\t\t\t"+"remoteTargetRefs[%s] = refBundle;\n\n"+
                "\t\t"+"} else ",
                remotePortMap.get (clientPort.port.portName)
            );
        }
    }

    /*
     * Throw an exception if the port is unknown.
     */
    result.format (
        "{\n"+
        "\t\t\t"+"throw new %s.ElementLinkException (\n"+
        "\t\t\t\t"+"\"Invalid remote client port '\"+ portName +\"'.\");\n"+
        "\t\t"+"}\n"+
        "\t"+"}\n\n",
        RUNTIME_PACKAGE
    );
}


protected void implementUnbindElRemotePort (StringBuilder output)
    throws TemplateProcessingException {

    Formatter result = new Formatter (output);

    /*
     * Method signature.
     */
    result.format (
        "\t"+"public final void unbindElRemotePort (String portName)\n"+
        "\t\t"+"throws %s.ElementLinkException {\n\n", RUNTIME_PACKAGE);

    /*
     * Lookup a REMOTE client port of given name and unbind it from the
     * remote target using.
     */
    result.format ("\t\t");
    for (ResolvedElementPort clientPort : ad.rei.port) {
        if (clientPort.port.type == PortType.REMOTE) {
            result.format ("if (\"%s\".equals (portName)) {\n", clientPort.port.portName);

            /*
             * Tell all the elements that subsume their remote client port
             * to this element remote client port to unbind the remote
             * port from the remote target.
             */
            for (ResolvedElementBinding bind : ad.rei.binding) {
                if (bind.isSubsumption (clientPort.port.portName)) {
                    String childPortName =
                        bind.getChildPortName ();
                    ResolvedElementInstance childElement =
                        bind.getChildElement ();
                    String childElementIndex =
                        elementInstMap.get (childElement);

                    result.format (
                        "\n"+
                        "\t\t\t"+"// inst.%s, type.%s, impl.%s\n"+
                        "\t\t\t"+"if (subElements[%s] instanceof %s.ElementRemoteClient) {\n"+
                        "\t\t\t\t"+"// unbind.%s.%s\n"+
                        "\t\t\t\t"+"((%s.ElementRemoteClient)\n"+
                        "\t\t\t\t\t"+"subElements[%s]).unbindElRemotePort(\"%s\");\n"+
```

```
                        "\t\t\t"+"}\n",
                        childElement.instanceName, childElement.elementType.typeName,
                            childElement.elementImplName,
                        childElementIndex, RUNTIME_PACKAGE,
                        childElement.instanceName, childPortName,
                        RUNTIME_PACKAGE,
                        childElementIndex, childPortName
                    );
                }
            }

            /*
             * Invalidate the remote reference bundle.
             */
            result.format (
                "\n"+
                "\t\t\t"+"remoteTargetRefs[%s] = null;\n\n"+
                "\t\t"+"} else ",
                remotePortMap.get (clientPort.port.portName)
            );
        }
    }

    /*
     * Throw an exception if the port is unknown.
     */
    result.format (
        "{\n"+
        "\t\t\t"+"throw new %s.ElementLinkException (\n"+
        "\t\t\t\t"+"\"Invalid remote client port '\"+ portName +\"'.\");\n"+
        "\t\t"+"}\n"+
        "\t"+"}\n\n",
        RUNTIME_PACKAGE
    );
}


protected void implementGetElRemoteTarget (StringBuilder output)
    throws TemplateProcessingException {

    Formatter result = new Formatter (output);

    /*
     * Method signature.
     */
    result.format (
        "\t"+"public final %s.RemoteRefBundle getElRemoteTarget (\n"+
        "\t\t\t"+"String portName)\n"+
        "\t\t"+"throws %s.ElementLinkException {\n\n",
        RUNTIME_PACKAGE, RUNTIME_PACKAGE
    );

    /*
     * Lookup a REMOTE client port of given name and return a bundle of
     * references to the remote target it is bound to.
     */
    result.format ("\t\t");
    for (ResolvedElementPort port : ad.rei.port) {
        if (port.port.type == PortType.REMOTE) {
            /*
             * Return the remote reference the port is bound to.
             */
            result.format (
                "if (\"%s\".equals (portName)) {\n"+
                "\t\t\t"+"return remoteTargetRefs[%s];\n\n"+
                "\t\t"+"} else ",
                port.port.portName,
                remotePortMap.get (port.port.portName)
            );
        }
    }

    /*
     * Throw an exception if the port is unknown.
     */
    result.format (
        "{\n"+
        "\t\t\t"+"throw new %s.ElementLinkException (\n"+
        "\t\t\t\t"+"\"Invalid remote client port '\"+ portName +\"'.\");\n"+
        "\t\t"+"}\n"+
        "\t"+"}\n\n",
        RUNTIME_PACKAGE
    );
}
```

```
/*************************************************************************
 * ReconfigurationHandler Methods
 *************************************************************************/

protected void implementInvalidateElPort (StringBuilder output)
    throws TemplateProcessingException {

    Formatter result = new Formatter (output);

    /*
     * Method signature.
     */
    result.format (
        "\t"+"public final void invalidateElPort(\n"+
        "\t\t\t"+"%s.ElementLocalServer element,\n"+
        "\t\t\t"+"String portName)\n"+
        "\t\t"+"throws %s.ReconfigurationException{\n\n"+
        "\t\t"+"try {\n\n",
        RUNTIME_PACKAGE, RUNTIME_PACKAGE
    );

    /*
     * Handle invalidate requests for local provided ports.
     *
     * If the invalidated port is local and part of a binding, query the
     * port for the local reference and pass it to the required side of the
     * binding. If the invalidated port is part of delegation, invalidate
     * the delegation port.
     *
     * So, go through all provided ports of all subelement instances,
     * determine the clients of these ports and generate code to rebind
     * the client ports to the provided ports.
     */
    result.format ("\t\t\t");
    for (ResolvedElementInstance inst : ad.rei.subElementInst) {
        /*
         * A map of clients of a PROVIDED port.
         */
        Map<ResolvedElementInstance, String> clients =
            new HashMap<ResolvedElementInstance, String> ();

        for (ResolvedElementPort port : inst.port) {

            /*
             * Skip ports that are not PROVIDED.
             */
            if (port.port.type != PortType.PROVIDED) {
                continue;
            }

            /*
             * Find client elements and ports that REQUIRE this port. For
             * this we have to find the bindings that have this element
             * and this particular port as the server part of the binding.
             * Then we can get the information about the client.
             */
            clients.clear ();
            for (ResolvedElementBinding bind : ad.rei.binding) {
                if (
                    bind.isBinding ()
                    &&
                    inst.instanceName.equals (bind.getServerElement ().instanceName)
                    &&
                    port.port.portName.equals (bind.getServerPortName ())
                ) {
                    clients.put (bind.getClientElement (), bind.getClientPortName ());
                }
            }

            /*
             * If there are no clients, the port is either unbound, or is
             * delegated to the parent element.
             *
             * TODO Handle the delegation...
             */
            if (clients.size () == 0) {
                continue;
            }

            /*
             * Get the reference from the provides port and
             * bind all the clients to it.
```

```
                 */
                ResolvedElementInstance serverElement = inst;
                String serverPortName = port.port.portName;
                String serverElementIndex = elementInstMap.get (serverElement);

                result.format (
                    "// inst %s, type %s, impl %s\n"+
                    "\t\t\t"+"if (element == subElements [%s] && \"%s\".equals (portName)) {\n"+
                    "\t\t\t\t"+"// lookup %s.%s\n"+
                    "\t\t\t\t"+"Object target = \n"+
                    "\t\t\t\t\t"+"((%s.ElementLocalServer) subElements [%s]).lookupElPort (\"%s\");\n\n",
                    serverElement.instanceName, serverElement.elementType.typeName,
                        serverElement.elementImplName,
                    serverElementIndex, serverPortName,
                    serverElement.instanceName, serverPortName,
                    RUNTIME_PACKAGE, serverElementIndex, serverPortName
                );

                /*
                 * Rebind all the clients...
                 */
                for (Entry<ResolvedElementInstance, String> client : clients.entrySet ()) {
                    ResolvedElementInstance clientElement = client.getKey ();
                    String clientPortName = client.getValue ();
                    String clientElementIndex = elementInstMap.get (clientElement);

                    result.format (
                        "\t\t\t"+"// rebind %s.%s\n"+
                        "\t\t\t"+"((%s.ElementLocalClient) subElements [%s]).bindElPort (\"%s\", target);\n\n",
                        clientElement.instanceName, clientPortName,
                        RUNTIME_PACKAGE, clientElementIndex, clientPortName
                    );
                }

                result.format ("\t\t"+"} else ");
            }
        }

        /*
         * Throw an exception if the port is unknown.
         *
         * Note: as a convenience, we throw an ElementLinkException even though
         * we should throw ReconfigurationException, because if we did not
         * generate any code (above), there will be no calls to bindElPort ()
         * and therefore nothing that could generate an ElementLinkException and
         * the generated code would fail to compile.
         */
        result.format (
            "{\n"+
            "\t\t\t"+"throw new %s.ElementLinkException (\n"+
            "\t\t\t\t"+ "\"Invalid reconfiguration request for port '\"+ portName +\"'.\");\n"+
            "\t\t"+"}\n\n",
            RUNTIME_PACKAGE
        );


        /*
         * Transform ElementLinkException to ReconfigurationException.
         */
        result.format (
            "\t"+"} catch (%s.ElementLinkException e) {\n"+
            "\t\t"+"throw new %s.ReconfigurationException (e);\n"+
            "\t"+"}\n"+
            "\t"+"}\n\n",
            RUNTIME_PACKAGE, RUNTIME_PACKAGE
        );
    }

}
```

Listing A.1: An example of a generator for a composite element

A.2 Generated element descriptor

The Java part of the proposed generator passes to the STRATEGO/XT part the XML descriptor of a connector element which should be generated. This process

is transparent to the user because the XML descriptor is automatically generated from the low-level configuration of the connector.

The listing below shows a descriptor of stub element, which is composed of two subelements.

```
<element>
  <!-- name of element. It is not compulsory. -->
  <name>LoggedClientUnit</stub>
  <!-- template file for element -->
  <template>compound_default.ellang</template>
  <!-- package for new element -->
  <package>generated.A00003</package>
  <!-- class name for generated element -->
  <classname>LoggedClientUnit</classname>
  <!-- list of ports -->
  <ports>
    <port name="call">
      <name>call</name>
      <signature>
        org.objectweb.dsrg.deployment.connector.generator.DemoIface
      </signature>
      <type>PROVIDED</type>
    </port>
    <port name="line">
      <name>call</name>
      <signature>
        org.objectweb.dsrg.deployment.connector.generator.DemoIface
      </signature>
      <type>REMOTE</type>
    </port>
  </ports>
  <!-- list of sub-elements -->
  <elements>
    <element>
      <name>stub</name>
      <class>generated.A00001.LocalStub</class>
    </element>
    <element>
      <name>logger</name>
      <class>generated.A00002.ConsoleLog</class>
    </element>
  </elements>
  <!-- description of bindings -->
  <bindings>
    <binding>
      <from>this.call</from>
      <to>logger.in</to>
    </binding>
    <binding>
      <from>logger.out</from>
      <to>stub.call</to>
    </binding>
    <binding>
      <from>this.line</from>
      <to>stub.line</to>
    </binding>
  </bindings>
  <!-- name of a folder where source codes are stored -->
  <source-folder>tests/generated/</source-folder>
  <!-- name of a folder where generated code will be stored -->
  <dest-folder>tests/generated/</dest-folder>
</element>
```

Listing A.2: XML descriptor of stub element

A.3 Element template presented in this thesis

This example shows a default template for a composite element, which is used for generation of the stub element described in *Section A.2 Generated element descriptor*.

In comparison with the preceding version of the generator, the STRATEGO based generator does not separate a connector element template into a static and a dynamic part. The logic of generation is located into the template itself.

```
/*
 * Name of package for generated class.
 */
package ${package}

/*
 * Some java specific imports.
 */
import org.objectweb.dsrg.deployment.connector.runtime.*;

/*
 * Meta declaration of connector element.
 *
 */
element compound_default {

 protected Element[] subElements;

 protected UnitReferenceBundle[] boundedToRemoteRef;

 public ${classname}() {
 }

 /*
  * Initilazation of element structure.
  */
 void initElStructure() throws ElementLinkException {
  subElements = new Element[${elements.element.size}];

  /* Create sub-elements */
  $set i = 0$;
  $foreach(ELEMENT in ${elements.element})$
   subElements[${i}] = new ${ELEMENT.class}();
   $set i = i + 1$
   /* remember ELEMENT index in array */
   $set ELEMENT.index = i$
  $end$

  /* create bindings */
  $foreach(BINDING in ${bindings.binding})$
   $if (BINDING.from.element.name != "this") $
    $if (BINDING.to.element.name != "this") $
     ((ElementLocalClient) subElements[${el[BINDING.to.element.name]}])
           .bindElPort("${BINDING.to.port}",
                 ((ElementLocalServer) subElements[${el[BINDING.from.element.name]}])
                    .lookupElPort("${BINDING.from.port}"));
    $end$
   $end$
  $end$

  /* Clients bounded to remote references - this part needs a number of remote ports */
  boundedToRemoteRef = new UnitReferenceBundle[${ports.port(type=REMOTE).size}];
  $set i = 0$
  $foreach(REMOTE_PORT in ${ports.port(type=REMOTE)})
   boundedToRemoteRef[${i}] = null;
   /* remember element index */
   $set ref[REMOTE_PORT.name] = i$
   $set i = i + 1$
  $end$
 }

 /* Implements interfaces */
 implements interface ElementLocalServer {
  public Object lookupElPort(String portName)
    throws ElementLinkException {
     /* this part needs number of PROVIDED ports and right index to subElements[] */
     $rforeach(PORT in ${ports.port(type=PROVIDED)} )$
      if ("${PORT.name}".equals(portName)) {
        return  ((ElementLocalServer) subElements[${el[PORT.boundedTo.element.name]}])
                                                .lookupElPort("${PORT.boundedTo.port}");
      } else
           $recpoint$
      $final$
           throw new  ElementLinkException("Invalid port '"+portName+"'.");
     $end$
  }
 }
```

```
implements interface ElementLocalClient {

 /* this needs number of REQUIRED ports and find right index of selected element */
 public void bindElPort(String portName, Object target) throws ElementLinkException {
  $rforeach(PORT in ${ports.port(type=REQUIRED)})$
   if ("${PORT.name}".equals(portName)) {
     ((ElementLocalClient) subElements[${el[PORT.boundedTo.element.name]}])
                                        .bindElPort("${PORT.boundedTo.port}", target);
   } else
        $recpoint$
   $final$
        throw new ElementLinkException("Invalid port '"+portName+"'.");
  $end$

 }

 public void unbindElPort(String portName) throws ElementLinkException {
  $rforeach(PORT in ${ports.port(type=REQUIRED)})$
   if ("${PORT.name}".equals(portName)) {
     ((ElementLocalClient) subElements[${el[PORT.boundedTo.element.name]}])
                                        .unbindElPort("${PORT.boundedTo.port}");
   } else
        $recpoint$
   $final$
        throw new ElementLinkException("Invalid port '"+portName+"'.");
  $end$
 }

}

implements interface ElementRemoteServer {

 public UnitReferenceBundle getElRemoteRefs(String portName)
   throws ElementLinkException {

   UnitReferenceBundle result = new UnitReferenceBundle();

   /* here we need number of REMOTE ports and right binding */
   $rforeach(PORT in ${ports.port(type=REMOTE)})$
   if ("${PORT.name}".equals(portName)) {
    if (subElements[${el[PORT.boundedTo.element.name]}] instanceof ElementRemoteServer) {
     result.addRefBundle(((ElementRemoteServer) subElements[${el[PORT.boundedTo.element.name]}]
                                                            .getElRemoteRefs("${PORT.boundedTo.port}")));
    }
   } else
        $recpoint$
   $final$
        throw new ElementLinkException("Invalid port '"+portName+"'.");
   $end$

   return result;
 }
}

implements interface ElementRemoteClient {

  public void bindElRemotePort(String portName, UnitReferenceBundle refBundle) throws ElementLinkException {

   /* here we need number of REMOTE ports and right binding */
   $rforeach(PORT in ${ports.port(type=REMOTE)})$
   if ("${PORT.name}".equals(portName)) {
    if (subElements[${el[PORT.boundedTo.element.name]}] instanceof ElementRemoteClient) {
     ((ElementRemoteClient) subElements[${ref[PORT.name]}])
                                        .bindElRemoteRefs("${PORT.boundedTo.port}", refBundle);
    }
    boundedToRemoteRef[${ref[PORT.name]}] = refBundle;
   } else
         $recpoint$
   $final$
         throw new ElementLinkException("Invalid port '"+portName+"'.");
   $end$
  }

  public void unbindElRemotePort(String portName) throws ElementLinkException {
   $rforeach(PORT in ${ports.port(type=REMOTE)})$
   if ("${PORT.name}".equals(portName)) {
    if (subElements[${el[PORT.boundedTo.element.name]}] instanceof ElementRemoteClient) {
     ((ElementRemoteClient) subElements[${ref[PORT.name]}]).unbindElRemoteRefs("${PORT.boundedTo.port}");
    }
    boundedToRemoteRef[${ref[PORT.name]}] = null;
   } else
         $recpoint$
   $final$
```

```
            throw new ElementLinkException("Invalid port '"+portName+"'.");
     $end$
    }

    public UnitReferenceBundle getElRemoteTarget(String portName) throws ElementLinkException {

     /* here we need number of REMOTE ports and right index of port into array boundedToRemoteRef */
     $rforeach(PORT in ${ports.port(type=REMOTE)})$
     if ("${PORT.name}".equals(portName)) {
      return boundedToRemoteRef[${ref[PORT.name]}];
     } else
            $recpoint$
     $final$
            throw new ElementLinkException("Invalid port '"+portName+"'.");
     $end$
    }


 }
}
```

Listing A.3: Source code of template element

A.4 Generated code

This listing shows code which is produced by the proposed connector element generator from the given element descriptor (see *Section A.2 Generated element descriptor*) and the template (see *Section A.3 Element template presented in this thesis*). Code is the same as code produced by the previous version of the generator.

```
package generated.A00000004;

import org.objectweb.dsrg.deployment.connector.runtime.*;

public class LoggedClientUnit implements ElementLocalServer,
                      ElementLocalClient,
                     ElementRemoteServer,
                     ElementRemoteClient {

  protected Element[] subElements;

  protected UnitReferenceBundle[] boundedToRemoteRef;

  public LoggedClientUnit() {
  }

  void initElStructure() throws ElementLinkException {
    subElements = new Element[2];
    subElements[0] = new generated.A00000002.LocalStub();
    subElements[1] = new generated.A00000005.ConsoleLog();

    ((ElementLocalClient) subElements[1]).bindElPort("out",
        ((ElementLocalServer) subElements[0]).lookupElPort("call"));

    boundedToRemoteRef = new UnitReferenceBundle[1];
    boundedToRemoteRef[0] = null;
  }

  public Object lookupElPort(String portName) throws ElementLinkException {

    if ("call".equals(portName)) {
      return ((ElementLocalServer) subElements[1]).lookupElPort("in");
    } else {
      throw new ElementLinkException("Invalid port '" + portName + "'.");
    }
  }

  public void bindElPort(String portName, Object target) throws ElementLinkException {
      throw new ElementLinkException("Invalid port '" + portName + "'.");
  }

  public void unbindElPort(String portName) throws ElementLinkException {
      throw new ElementLinkException("Invalid port '" + portName + "'.");
```

```
  }


  public UnitReferenceBundle getElRemoteRefs(String portName)
      throws ElementLinkException {

    UnitReferenceBundle result = new UnitReferenceBundle();

    if ("line".equals(portName)) {

      if (subElements[0] instanceof ElementRemoteServer) {
        result.addRefBundle(((ElementRemoteServer) subElements[0])
            .getElRemoteRefs("line"));
      }
    } else {
      throw new ElementLinkException("Invalid port '" + portName + "'.");
    }

    return result;
  }

  public void bindElRemotePort(String portName, UnitReferenceBundle refBundle)
      throws ElementLinkException {

    if ("line".equals(portName)) {

      if (subElements[0] instanceof ElementRemoteClient) {
        ((ElementRemoteClient) subElements[0]).bindElRemotePort("line",
            refBundle);
      }
      boundedToRemoteRef[0] = refBundle;
    } else {
      throw new ElementLinkException("Invalid port '" + portName + "'.");
    }
  }

  public void unbindElRemotePort(String portName) throws ElementLinkException {

    if ("line".equals(portName)) {

      if (subElements[0] instanceof ElementRemoteClient) {
        ((ElementRemoteClient) subElements[0])
            .unbindElRemotePort("line");
      }
      boundedToRemoteRef[0] = null;
    } else {
      throw new ElementLinkException("Invalid port '" + portName + "'.");
    }
  }

  public UnitReferenceBundle getElRemoteTarget(String portName)
      throws ElementLinkException {

    if ("line".equals(portName)) {
      return boundedToRemoteRef[0];
    } else {
      throw new ElementLinkException("Invalid port '" + portName + "'.");
    }
  }

}
```

Listing A.4: Generated element Java code

Appendix B

Content of attached CD ROM

This thesis is accompanied by the CD ROM containing binaries and source code of the prototype implementation. The CD ROM is organized as follows:

`/README.TXT`
: Brief description of the content of the CD ROM.

`/doc/`
: Electronic version of this thesis.

`/src/congen/java/`
: Source codes of the existing connector generator extended by a bridge connecting it with the STRATEGO part.

`/src/congen/stratego/`
: Source codes of the STRATEGO based implementation of the connector generator.

`/examples/`
: Examples of connectors elements written in the ELLANG-J language

`/prerequisites/`
: Software prerequisites of the prototype: StrategoXT v0.16, ATerm library v2.4.2